BACHARELADO EM
CIÊNCIA DA COMPUTAÇÃO

<0ficinas 1ntegradoras/>

ANO 1
2026

INSTITUTO FEDERAL
Goiano

Campus
Rio Verde

Dados Internacionais de Catalogação na Publicação (CIP)
Sistema Integrado de Bibliotecas (SIBI) – Instituto Federal Goiano

O48 Oficinas integradoras (1. : 2026: Rio Verde, GO)

Oficinas integradoras [Recurso eletrônico] / Organizadores: Arthur Barbosa Lunas ... [et al.] -- Rio Verde, GO: IF Goiano, 2026.
214 p.: il.: color.

Livro referente a oficinas de extensão do curso de Bacharelado em Ciências da Computação do IF Goiano - Campus Rio Verde - GO.

1. Ciência e tecnologia. 2.Computação. 3. Extensão. I. Lunas, Arthur Barbosa. II. Santos, Camila Gabriele Figueredo dos. III. Souza, Daniel Alves de. IV. Giovannetti, Eduardo Augusto Duarte de Castro. V. Goulart, Eduardo Augusto Oliveira. VI. Bartasson, Guilherme Carlos Freitas. VII. Souza, Helder Oliveira Gomes de. VIII. Pimentel, Josué Farias. IX. Soares, Kaike Leles Lamonier. X. Silva, Marcos Santos da. XI. Cazon, Paulo Felipe Candia. XII. Cardoso, Pedro Lucas Freitas. XIII. Souza, Sidiney Barbosa de. XIV. Amaral, Stephanny Kauane Oliveira. XV. Título.

CDU 004

Responsável: Hevellin Estrela - Bibliotecária-Documentalista CRB-1 n°2453

SERVIÇO PÚBLICO FEDERAL
MINISTÉRIO DA EDUCAÇÃO
SECRETARIA DE EDUCAÇÃO PROFISSIONAL E TECNOLÓGICA
INSTITUTO FEDERAL DE EDUCAÇÃO, CIÊNCIA E TECNOLOGIA GOIANO

TERMO DE AUTORIZAÇÃO PARA DISPONIBILIZAR PRODUÇÃO TÉCNICA NO REPOSITÓRIO INSTITUCIONAL DO IF GOIANO

Repositório Institucional do IF Goiano - RIIF Goiano Sistema Integrado de Bibliotecas

- Profissional de Educação do IF Goiano -

Com base no disposto na Lei Federal nº 9.610/98, e manual sobre a Produção Técnica, publicado pela DAV/CAPES/MEC*, AUTORIZO o Instituto Federal de Educação, Ciência e Tecnologia Goiano, a disponibilizar gratuitamente o documento no Repositório Institucional do IF Goiano (RIIF Goiano), sem ressarcimento de direitos autorais, conforme permissão assinada eletronicamente abaixo, em formato digital para fins de leitura, download e impressão, a título de divulgação da produção técnico-científica no IF Goiano.

Identificação da Produção Técnica – DAV/CAPES

[ ] Editoria [ ] Material Didático

[ ] Curso de Formação Profissional [X] Projetos de Extensão à Comunidade

[ ] Relatório Técnico Conclusivo [ ] Atividade Técnica/Tecnológica

[ ] Disseminação do Conhecimento Técnico/Tecnológico [ ] Produto Bibliográfico

[ ] Outras Produções Técnicas - Tipo:

Nome Completo do Autor/a:
Arthur Barbosa Lunas, Camila Gabriele Figueredo dos Santos, Daniel Alves de Sousa, Eduardo Augusto Duarte de Castro Giovannetti, Eduardo Augusto Oliveira Goulart, Guilherme Carlos Freitas Bartasson, Helder Oliveira Gomes De Souza, Josué Farias Pimentel, Kaike Leles Lamonier Soares, Marcos Santos da Silva, Paulo Felipe Candia Cazon, Pedro Lucas Freitas Cardoso, Sidiney Barbosa de Souza, Stephanny Kauane Oliveira Amaral

Matrícula:
2023102201940268, 2023102201940101, 2023102201940020, 2023102201940063, 2023102201940241, 2023102201940080, 2024202192010004, 2023102201940306, 2023102201940233, 2024102192010016, 2023102201940098, 2023102201940039, 2023102201940071, 2023102201940195

Título do Trabalho: Oficinas integradoras - 2026 - Ano 1

Restrições de Acesso ao Documento

Documento confidencial: [X] Não [ ] Sim

Justifique: _____

Informe a data que poderá ser disponibilizado no RIIF Goiano: 10 / 07 / 2026

O documento está sujeito a registro de patente? [ ] Sim [X] Não

O documento pode vir a ser publicado como livro e/ou artigo? [ ] Sim [X] Não

DECLARAÇÃO DE DISTRIBUIÇÃO NÃO-EXCLUSIVA

O/A referido/a docente e/ou autor/a declara que:

1 - o documento é seu trabalho original, detém os direitos autorais da produção técnica e não infringe os direitos de qualquer outra pessoa ou entidade;

2 - obteve autorização de quaisquer materiais inclusos no documento do qual não detém os direitos de autor/a, para conceder ao Instituto Federal de Educação, Ciência e Tecnologia Goiano os direitos requeridos e que este material cujos direitos autorais são de terceiros, estão claramente identificados e reconhecidos no texto ou conteúdo do documento entregue;

3 - cumpriu quaisquer obrigações exigidas por contrato ou acordo, caso o documento entregue seja baseado em trabalho financiado ou apoiado por outra instituição que não o Instituto Federal de Educação, Ciência e Tecnologia Goiano.

Rio Verde, 1 de julho de 2026.

(Assinado Eletronicamente)

Arthur Barbosa Lunas, Camila Gabriele Figueredo dos Santos, Daniel Alves de Sousa, Eduardo Augusto Duarte de Castro Giovannetti, Eduardo Augusto Oliveira Goulart, Guilherme Carlos Freitas Bartasson, Helder Oliveira Gomes De Souza, Josué Farias Pimentel, Kaike Leles Lamonier Soares, Marcos Santos da silva, Paulo Felipe Candia Cazon, Pedro Lucas Freitas Cardoso, Sidiney Barbosa de Souza, Stephanny Kauane Oliveira Amaral
(Autores e/ou Detentores dos Direitos Autorais)

(Assinado Eletronicamente)

Douglas Cedrim Oliveira (Orientador)

1058004

(Assinatura do orientador)

Documento assinado eletronicamente por:

- **Douglas Cedrim Oliveira**, **PROFESSOR ENS BASICO TECN TECNOLOGICO** , em 01/07/2026 20:08:41.
- **Camila Gabriele Figueredo dos Santos**, **2023102201940101 - Discente**, em 01/07/2026 20:16:59.
- **Paulo Felipe Candia Cazon**, **2023102201940098 - Discente**, em 01/07/2026 20:24:24.
- **Guilherme Carlos Freitas Bartasson**, **2023102201940080 - Discente**, em 01/07/2026 21:00:01.
- **Arthur Barbosa Lunas**, **2023102201940268 - Discente**, em 02/07/2026 00:49:01.
- **Daniel Alves de Sousa**, **2023102201940020 - Discente**, em 02/07/2026 08:46:24.
- **Eduardo Augusto Duarte de Castro Giovannetti**, **2023102201940063 - Discente**, em 02/07/2026 11:43:56.
- **Kaike Leles Lamonier Soares**, **2023102201940233 - Discente**, em 02/07/2026 13:34:10.
- **Stephanny Kauane Lima Oliveira**, **2023102201940195 - Discente**, em 02/07/2026 13:42:47.
- **Eduardo Augusto Oliveira Goulart**, **2023102201940241 - Discente**, em 02/07/2026 15:14:00.
- **Pedro Lucas Freitas Cardoso** , **2023102201940039 - Discente**, em 02/07/2026 16:49:16.
- **Sidiney Barbosa de Souza**, **2023102201940071 - Discente**, em 02/07/2026 21:23:49.
- **Josué Farias Pimentel**, **2023102201940306 - Discente**, em 04/07/2026 08:45:30.
- **Helder Oliveira Gomes De Souza**, **2024202192010004 - Discente**, em 30/07/2026 20:19:30.
- **Marcos Santos da silva**, **2024102192010016 - Discente**, em 03/08/2026 22:37:10.

Este documento foi emitido pelo SUAP em 01/07/2026. Para comprovar sua autenticidade, faça a leitura do QRCode ao lado ou acesse https://suap.ifgoiano.edu.br/autenticar-documento/ e forneça os dados abaixo:

Código Verificador: 836934
Código de Autenticação: 101020b78e

INSTITUTO FEDERAL GOIANO
Campus Rio Verde
Rodovia Sul Goiana, Km 01, Zona Rural, 01, Zona Rural, RIO VERDE / GO, CEP 75901-970
(64) 3624-1000

INSTITUTO FEDERAL GOIANO - CAMPUS RIO VERDE
BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

Coordenação do curso
Heverton Barros de Macêdo (Coordenador)
Douglas Cedrim (Vice-coordenador)

Orientação e editoração
Douglas Cedrim

Organização
Arthur Barbosa Lunas
Camila Gabriele Figueredo dos Santos
Daniel Alves de Sousa
Douglas Cedrim
Eduardo Augusto Duarte de Castro Giovannetti
Eduardo Augusto Oliveira Goulart
Guilherme Carlos Freitas Bartasson
Helder Oliveira Gomes de Souza
Josué Farias Pimentel
Kaike Leles Lamonier Soares
Marcos Santos da Silva
Paulo Felipe Candia Cazon
Pedro Lucas Freitas Cardoso
Sidiney Barbosa de Souza
Stephanny Kauane Oliveira Amaral

MENSAGEM DOS ORGANIZADORES

A ideia das Oficinas Integradoras surgiu inicialmente dentro da disciplina de Projetos Integradores 3, no bacharelado em Ciência da Computação, que tinha como parte avaliativa a elaboração de minicursos sobre temas ligados à computação.

Essa ideia evoluiu e se tornou um evento de extensão, organizado pelos discentes da disciplina, que são discentes do curso de bacharelado em Ciência da Computação, com orientação por professores do curso. Consiste em um evento realizado no Instituto Federal de Educação, Ciência e Tecnologia Goiano Campus Rio Verde - IF Goiano RV, no prédio da computação. O objetivo principal é difundir parte do conhecimento adquirido ao longo do curso de computação para a comunidade, em forma de minicursos, valorizando o conhecimento científico, tecnológico e possibilitando sua difusão.

Esperamos que este material contribua com sua formação!

Conteúdo

Minicurso 1

Introdução à construção de interfaces gráficas utilizando Raylib

Pedro Lucas Freitas Cardoso e Josué Farias Pimentel

Resumo

Este minicurso apresenta uma introdução ao desenvolvimento de interfaces gráficas utilizando a linguagem de programação C e a biblioteca Raylib. O objetivo é demonstrar os conceitos básicos de criação de janelas, renderização de elementos gráficos e interação com o usuário, permitindo que os participantes desenvolvam suas primeiras aplicações gráficas de forma simples e prática. São abordados o loop de janela, controle de quadros por segundo, funções de desenho, leitura de teclado e mouse, e o processo de compilação multiplataforma.

1.1 Introdução

Este minicurso tem como objetivo introduzir o desenvolvimento de aplicações com interface gráfica utilizando a linguagem de programação C.

A Raylib [Raylib Technologies, 2026] é uma biblioteca gráfica multiplataforma e de código aberto (*open source*) desenvolvida para facilitar a criação de jogos e aplicações 2D e 3D. Sua principal característica é a simplicidade de uso, permitindo que iniciantes desenvolvam interfaces gráficas e aplicações visuais sem a complexidade normalmente encontrada em bibliotecas gráficas mais avançadas.

Além disso, a Raylib oferece suporte a diferentes sistemas operacionais, como Windows, Linux e macOS. Dessa forma, grande parte do código-fonte pode ser reutilizada entre plataformas, sendo necessárias apenas pequenas adaptações no processo de compilação. Essa característica torna a biblioteca uma excelente opção para quem está tendo o primeiro contato com o desenvolvimento de interfaces gráficas (*Graphical User Interfaces* – GUIs).

O desenvolvimento de jogos é uma atividade que estimula a aplicação prática de diversos conceitos de programação, como estruturas de dados, lógica, matemática, eventos e manipulação gráfica. Nesse contexto, a utilização de uma biblioteca simples e acessível como a Raylib contribui para tornar o processo de aprendizagem mais dinâmico e motivador [Albuquerque et al., 2025].

1.2 Público-alvo

O público-alvo deste minicurso são estudantes e entusiastas interessados na criação de interfaces gráficas. Para acompanhar o conteúdo, é necessário ter conhecimentos básicos da linguagem C, como a criação e manipulação de variáveis, vetores e matrizes, noções básicas de ponteiros e a criação de funções.

1.3 Pré-requisitos

Como pré-requisito, é necessário que o aluno possua a bagagem de conhecimento citada na seção anterior. Também será preciso um computador com Linux [Linux Foundation, 2026] ou Windows [Microsoft, 2026] instalado, um compilador C, de preferência o GCC, e a biblioteca Raylib devidamente instalada. Recomenda-se ainda o uso de um editor de código ou ambiente de desenvolvimento, como VS Code [Visual Studio Code, 2026] ou outro de preferência do aluno.

É desejável que o participante tenha familiaridade básica com o uso do terminal ou prompt de comando para compilar e executar programas em C. Como tanto a linguagem de programação quanto a biblioteca são leves, não será necessário um computador com alto poder de processamento para acompanhar o minicurso.

1.4 Roteiro

O minicurso conta com 1h e 30m de duração e seguirá o roteiro contido na Tabela 1.1.

1.5 Estrutura básica de um programa Raylib

Todo programa em Raylib segue sempre o mesmo esqueleto: inicializa a janela, executa o loop principal enquanto a janela estiver aberta e, ao terminar, libera os recursos.

Tabela 1.1: Roteiro do minicurso com a duração aproximada de cada conteúdo.

Tópico	Conteúdo	Duração
Introdução	Apresentação da Raylib, principais características da biblioteca e exemplos de aplicações desenvolvidas com ela.	10 minutos
Configuração do Ambiente	Instalação da biblioteca Raylib, compilação de um exemplo simples e preparação do ambiente de desenvolvimento.	10 minutos
Estrutura de um Programa Raylib	Estrutura básica de um programa utilizando Raylib: criação da janela, loop principal, funções de renderização e encerramento da aplicação.	15 minutos
Desenho de Elementos Gráficos	Desenho de formas básicas e textos na tela utilizando funções como `DrawText`, `DrawRectangle` e `DrawCircle`. Uso de cores e atualização da tela.	20 minutos
Entrada do Usuário	Captura de eventos de teclado e mouse utilizando funções da Raylib, permitindo a interação do usuário com a aplicação.	15 minutos
Exemplo Prático	Desenvolvimento de uma pequena aplicação interativa utilizando os conceitos apresentados ao longo do minicurso.	20 minutos

Fonte: Autoria Própria.

Código-fonte 1 Código mínimo para criar uma janela.

```c
#include "raylib.h"

int main(void) {
    // 1. Inicializa a janela (largura, altura, titulo)
    InitWindow(800, 600, "Minha Janela");

    // 2. Define quantos quadros por segundo renderizar
    SetTargetFPS(60);

    // 3. Loop principal -- repete enquanto a janela nao for fechada
    while (!WindowShouldClose()) {

        // -- Logica do jogo/app vai aqui --

        BeginDrawing();
        ClearBackground(RAYWHITE); // limpa a tela

        EndDrawing();
    }

    // 4. Fecha a janela e libera recursos
    CloseWindow();
    return 0;
}
```

Fonte: Autoria própria.

Regra de ouro: todo o desenho deve acontecer entre `BeginDrawing()` e `EndDrawing()`. Chamadas de lógica (movimentação, colisão, entrada) ficam fora

desse bloco, antes de `BeginDrawing()`.

O sistema de coordenadas tem origem $(0, 0)$ no canto superior esquerdo da janela. O eixo X cresce para a direita e o eixo Y cresce para baixo.

1.6 Controlando os quadros por segundo — `SetTargetFPS`

A função `SetTargetFPS(fps)` instrui a Raylib a manter uma taxa de atualização próxima ao valor informado. Internamente, ela calcula quanto tempo cada frame deveria durar e introduz um pequeno atraso ao fim de `EndDrawing()` para não ultrapassar esse limite.

Código-fonte 2 Definição de limite de quadros por segundo da janela.

```
1 SetTargetFPS(60); // ~60 frames por segundo (padrao para jogos)
2 SetTargetFPS(30); // ~30 fps -- economiza CPU em apps simples
3 SetTargetFPS(144); // ~144 fps -- para monitores de alta taxa
4 SetTargetFPS(0); // sem limite -- roda o mais rapido possivel
```

Fonte: Autoria própria.

Quando chamar? Sempre antes do loop principal, logo após `InitWindow()`. É possível chamar novamente dentro do loop para alterar o FPS dinamicamente durante a execução.

Por que o FPS importa?

A Tabela 1.2 mostra o impacto prático da escolha de FPS.

Tabela 1.2: Comparação entre diferentes valores de FPS alvo.

FPS alvo	Tempo por frame	Uso típico	Carga de CPU
30	≈33 ms	Apps simples, jogos casuais	Baixa
60	≈16 ms	Padrão para jogos	Média
120	≈8 ms	Jogos de ação rápida	Alta
0	máximo possível	Benchmarks, testes	Máxima

Fonte: Autoria própria.

Atenção: `SetTargetFPS` é um limite máximo, não uma garantia. Se a lógica do jogo demorar mais do que o tempo de um frame, o FPS real cairá abaixo do alvo. Monitore com `DrawFPS(x, y)` durante o desenvolvimento.

Para exibir ou ler o FPS em tempo real:

Código-fonte 3 Controle de Quadros por Segundo.

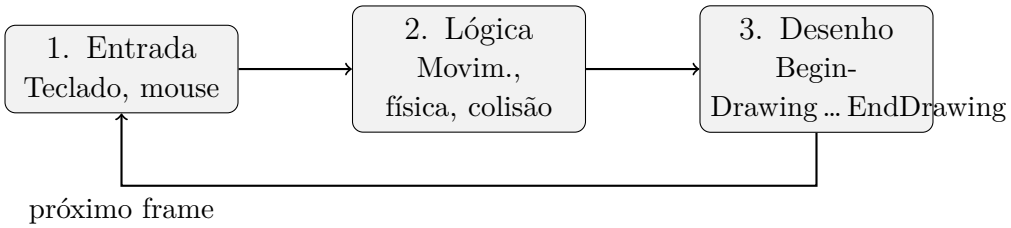
```
// Exibe o FPS atual na posicao (10, 10) da tela
DrawFPS(10, 10);

// Ou leia o valor para usar na logica do programa
int fpsAtual = GetFPS();
```

Fonte: Autoria própria.

1.7 O loop da janela em detalhe

O loop principal é o coração de qualquer programa Raylib. Ele executa continuamente, frame a frame, até que o usuário feche a janela. Cada iteração é chamada de *frame* e segue sempre a mesma sequência de três etapas, ilustradas na Figura 1.1.

Figura 1.1: As três etapas de cada frame no loop principal da Raylib.



Fonte: Autoria própria.

Código-fonte 4 Divisão de etapas de um programa usando raylib.

```
while (!WindowShouldClose()) {

    // ===== ETAPA 1: ENTRADA =====
    if (IsKeyPressed(KEY_ESCAPE)) break;

    // ===== ETAPA 2: LOGICA =====
    float delta = GetFrameTime(); // tempo do ultimo frame em segundos
    posX += velocidade * delta;

    // ===== ETAPA 3: DESENHO =====
    BeginDrawing();
    ClearBackground(RAYWHITE);
    DrawCircle((int)posX, 300, 30, BLUE);
    DrawFPS(10, 10);
    EndDrawing();
}
```

Fonte: Autoria própria.

A função `WindowShouldClose()`

Essa função retorna `true` em duas situações: quando o usuário clica no botão fechar (**X**) da janela ou quando pressiona `ESC`. O comportamento do ESC pode ser reconfigurado:

Código-fonte 5 Definição do tecla para fechar a janela.

```
// Desabilita o ESC como tecla de fechamento
SetExitKey(KEY_NULL);

// Ou redefine para outra tecla
SetExitKey(KEY_Q);
```

Fonte: Autoria própria.

Tempo entre frames — `GetFrameTime()`

A cada frame, o tempo decorrido desde o frame anterior varia levemente. `GetFrameTime()` retorna esse intervalo em segundos e é fundamental para criar movimentos suaves e independentes do FPS.

Código-fonte 6 Controle de movimento baseado no FPS.

```
float delta = GetFrameTime(); // tipicamente ~0.016 a 60 fps

// SEM delta time -- o objeto anda 5 pixels por frame
// Em 60 fps = 300 px/s | Em 30 fps = 150 px/s (INCONSISTENTE!)
posX += 5.0f;

// COM delta time -- o objeto anda 300 pixels por SEGUNDO, sempre
// Em 60 fps = 300 px/s | Em 30 fps = 300 px/s (CONSISTENTE!)
posX += 300.0f * delta;
```

Fonte: Autoria própria.

Regra prática: sempre multiplique velocidades e acelerações por `GetFrameTime()`. Assim o comportamento do programa é idêntico em qualquer máquina, independentemente do hardware ou do FPS atingido.

1.8 Funções principais de desenho

Texto

Código-fonte 7 Desenhando texto com `DrawText`.

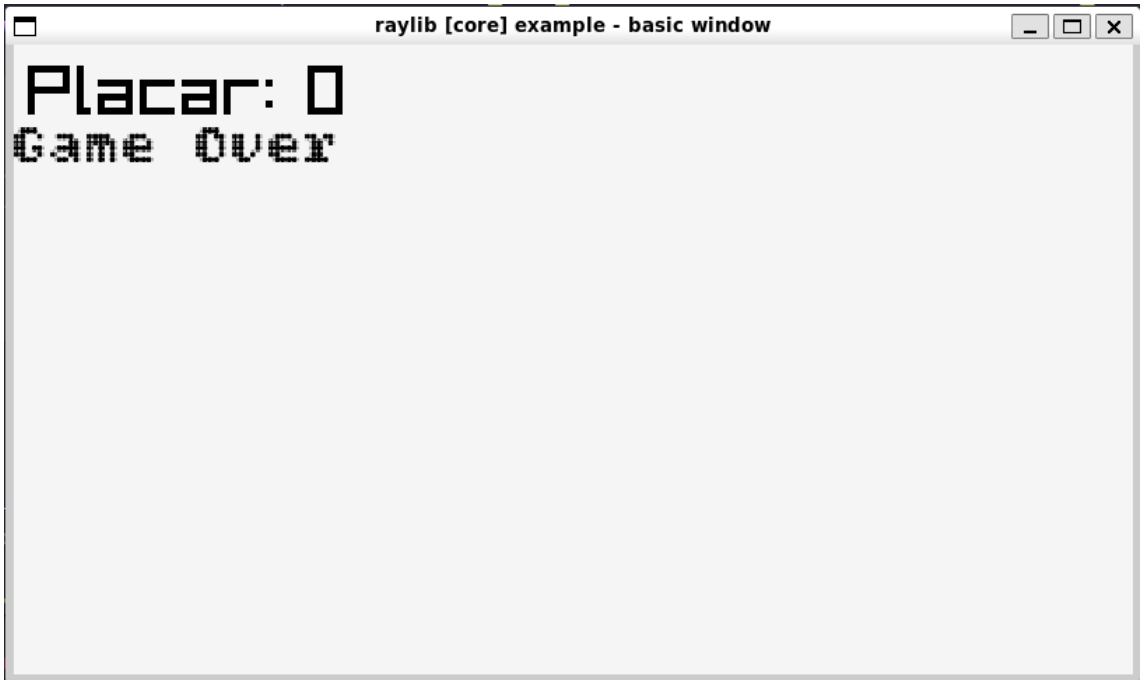
```
1  Font font = LoadFont("fonts/Bitcount_Grid_Double/font.ttf");
2  while (!WindowShouldClose())
3  {
4      BeginDrawing();
5      ClearBackground(RAYWHITE);
6      // DrawText(texto, posX, posY, tamanhoFonte, cor)
7      DrawText("Placar: 0", 10, 10, 50, BLACK);
8
9      // DrawTextEx(Font font, const char *text, Vector2 position,float
   ↪  fontSize, float spacing, Color tint);
10     DrawTextEx(font, "Game Over", (Vector2){0, 50}, 50, 1, BLACK);
11     EndDrawing();
12 }
```

Fonte: Autoria própria.

Como apresentado no Código-fonte 7, as funções DrawText e DrawTextEx são utilizadas para exibir texto na janela. A função DrawText é a mais simples de usar, pois não requer a importação de fontes externas, utilizando a fonte padrão da raylib.

Por outro lado, a função DrawTextEx oferece maior flexibilidade e contribui para um resultado mais profissional, permitindo ao desenvolvedor utilizar fontes personalizadas e estilizadas.

Figura 1.2: Renderização do algoritmo de exemplo 7.



Fonte: Autoria própria.

A Figura 1.2 apresenta o resultado da renderização do Código-fonte 7, evidenciando o uso das funções `DrawText` e `DrawTextEx`. No topo da janela, observa-se o texto ”Placar: 0”renderizado com a fonte padrão da raylib, demonstrando a simplicidade de uso da função `DrawText`. Já o texto ”Game Over”é exibido utilizando uma fonte externa carregada previamente, por meio da função `DrawTextEx`, o que proporciona maior controle estético e personalização. Esse exemplo ilustra como diferentes abordagens de renderização de texto podem ser combinadas em uma mesma aplicação para atender tanto a simplicidade quanto a customização visual.

Código-fonte 8 Estrutura de vetor 2d (Vector2).

```
1 typedef struct{
2     float x;
3     float y;
4 }Vector2;
```

Fonte: Autoria própria.

A estrutura Vector2, apresentada no Código-fonte 8, é amplamente utilizada para representar posições bidimensionais de forma encapsulada. Em vez de manipular separadamente as coordenadas x e y, essa estrutura permite agrupar ambos os valores em uma única entidade, tornando o código mais organizado, legível e fácil de manter.

Código-fonte 9 Importação de fontes externas.

```
1 //Carrega a fonte na GPU
2 Font font1 = LoadFont("/pasta_com_a_fonte/font.tff");
3
4 //Antes de fechar a janela eh necessario dar unload nas fontes
5 Unload(font1);
```

Fonte: Autoria própria.

Como mostrado no Código-fonte 9, é possível importar fontes externas para serem utilizadas na renderização de texto. A função LoadFont é responsável por carregar o arquivo de fonte e disponibilizá-lo na GPU, permitindo seu uso em funções como DrawTextEx.

Além disso, é importante liberar os recursos alocados após o uso da fonte. Para isso, utiliza-se a função Unload, que remove a fonte da memória da GPU antes do encerramento do programa, evitando vazamentos de memória e garantindo uma melhor gestão de recursos.

Retângulos

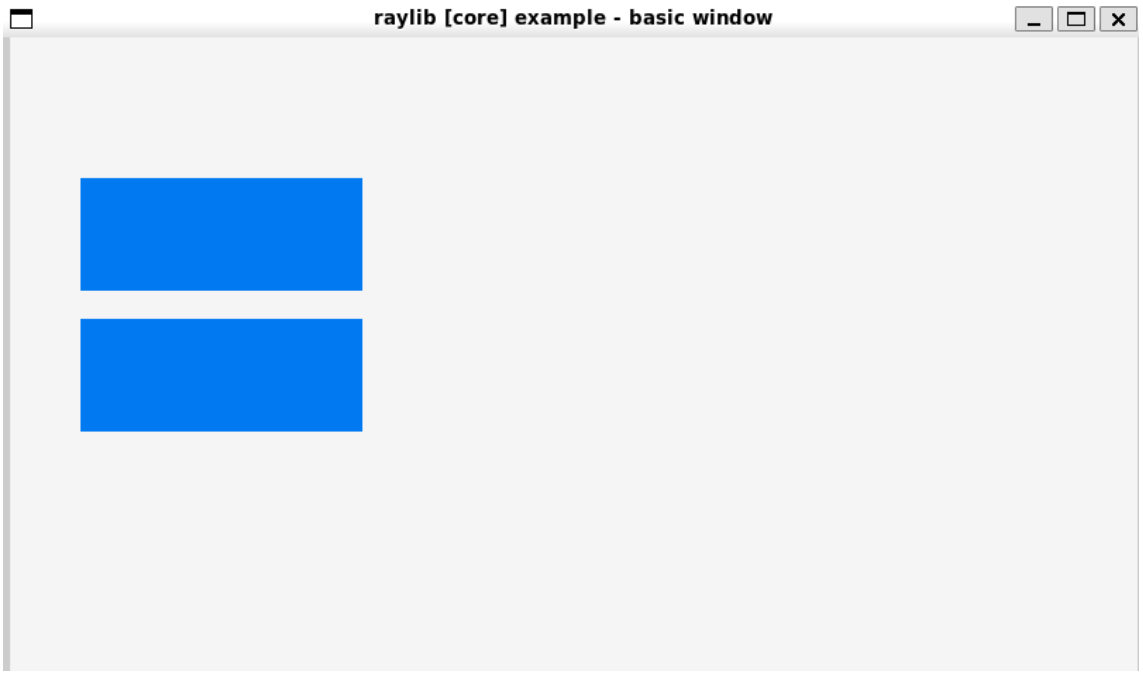
Código-fonte 10 Desenhando retângulos preenchidos.

```
1 // DrawRectangle(float x, float y, float width, float height, Color
  ↪  color)
2 DrawRectangle(50, 100, 200, 80, BLUE);
3
4 Rectangle rec1 = {50, 200, 200, 80};
5 //DrawRectangleRec(Rectalgle rec, Color color)
6 DrawRectangleRec(rec1, BLUE);
```

Fonte: Autoria própria.

No Código-fonte 10, são apresentadas funções utilizadas para desenhar retângulos preenchidos na tela. A função `DrawRectangle` permite definir manualmente a posição (`x`, `y`) e as dimensões (`width`, `height`), além da cor do retângulo. Já a função `DrawRectangleRec` utiliza uma estrutura do tipo `Rectangle`, facilitando o uso quando os dados do retângulo já estão organizados em uma única variável. Ambas as funções preenchem completamente a área do retângulo com a cor especificada.

Figura 1.3: Figura da renderização do exemplo do Código-fonte 10.



Fonte: Autoria própria.

A saída esperada para o exemplo 10 pode ser visualizada na imagem 1.3, onde se pode ver os dois retangulos azuis.

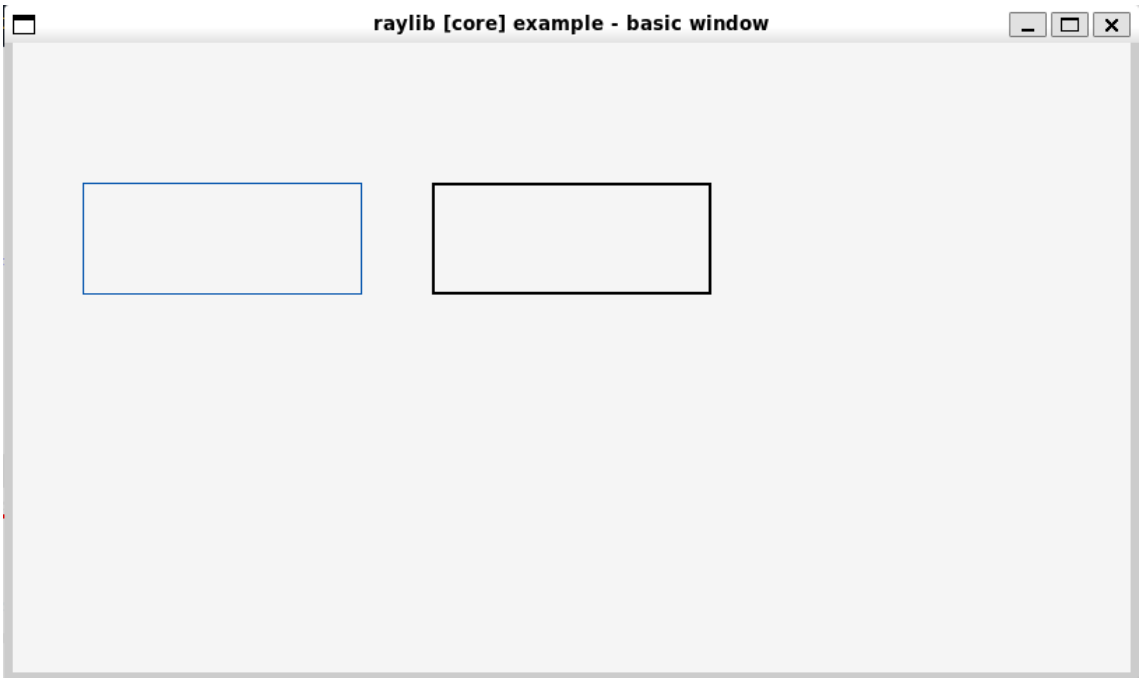
Código-fonte 11 Desenhando retângulos vazados.

```
1 // DrawRectangleLines(float x, float y, float width, float height, Color
  ↪  color)
2 DrawRectangleLines(50, 100, 200, 80, DARKBLUE);
3
4 Rectangle rec1 = { 300, 100, 200, 80 };
5 // DrawRectangleLinesEx(Rectangle rec, float lineThick, Color color);
6 DrawRectangleLinesEx(rec1, 2, BLACK);
```

Fonte: Autoria própria.

No Código-fonte 11, são demonstradas funções para desenhar apenas o contorno de retângulos, ou seja, sem preenchimento interno. A função `DrawRectangleLines` permite especificar diretamente a posição e as dimensões, desenhando as bordas com uma espessura padrão. Por outro lado, `DrawRectangleLinesEx` oferece maior controle ao permitir definir explicitamente a espessura da linha, além de utilizar uma estrutura `Rectangle` para representar o retângulo. Essas funções são úteis para destacar áreas ou criar interfaces visuais mais detalhadas.

Figura 1.4: Renderização do algoritmo de exemplo 11.



Fonte: Autoria própria.

A Figura 1.4 apresenta o resultado da execução do código, evidenciando dois retângulos desenhados apenas com suas bordas. O primeiro utiliza a função básica com espessura padrão, de apenas 1 pixel, enquanto o segundo apresenta uma espessura de linha personalizada de 2 pixels. Esse tipo de representação é útil para delimitação de áreas, criação de interfaces e destaque de elementos gráficos sem preenchimento interno.

Código-fonte 12 Estrutura Rectangle.

```
typedef struct{
    float x;
    float y;
    float width;
    float height;
} Rectangle
```

Fonte: Autoria própria.

A estrutura `Rectangle`, apresentada no Código-fonte 12, é utilizada para representar retângulos em um plano bidimensional. Ela armazena a posição do retângulo por meio das coordenadas `x` e `y`, além de suas dimensões, definidas por `width` (largura) e `height` (altura). Essa forma de organização permite manipular retângulos de maneira mais prática e estruturada, sendo amplamente utilizada em funções gráficas para desenho, colisão e posicionamento de elementos na tela.

Círculos

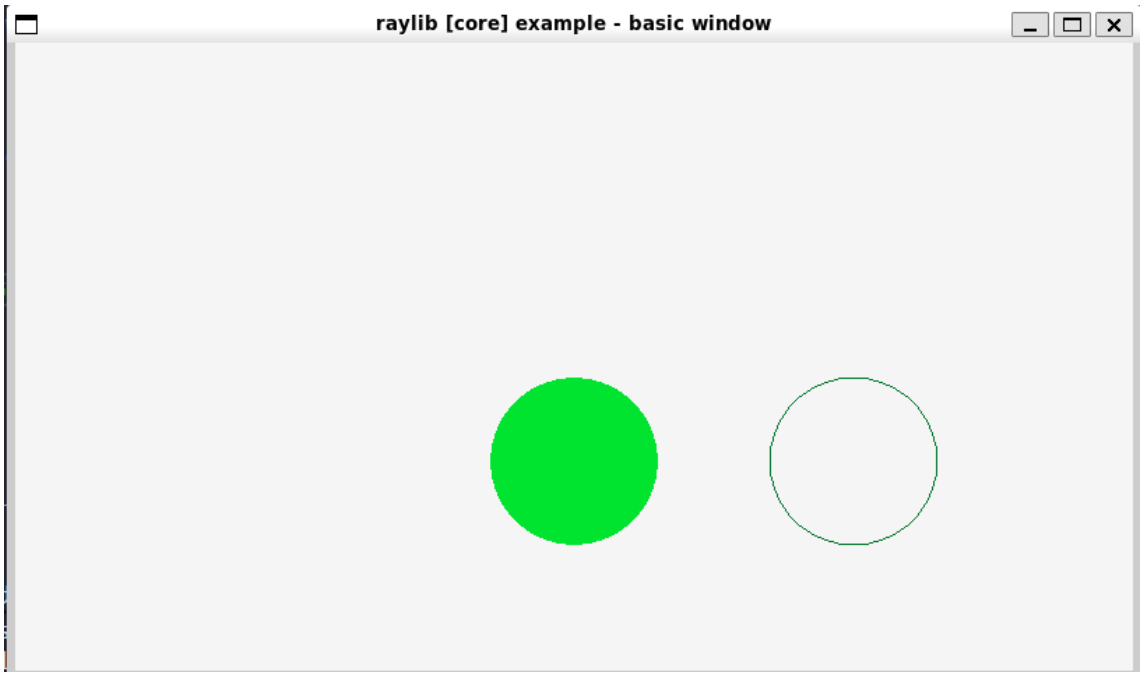
Código-fonte 13 Desenhando círculos preenchidos e vazados.

```
// DrawCircle(centroX, centroY, raio, cor)
DrawCircle(400, 300, 60, GREEN);

// Versao sem preenchimento
DrawCircleLines(600, 300, 60, DARKGREEN);
```

Fonte: Autoria própria.

O trecho apresentado no Código-fonte 13 utiliza as funções `DrawCircle` e `DrawCircleLines` para a construção de círculos na tela. A primeira função desenha um círculo preenchido, permitindo definir sua posição por meio das coordenadas do centro, o raio e a cor desejada. Já a segunda função desenha apenas o contorno do círculo, sem preenchimento interno, sendo útil para destacar elementos visuais ou representar formas vazadas.

Figura 1.5: Renderização dos círculos apresentados no algoritmo de exemplo.



Fonte: Autoria própria.

A Figura 1.5 apresenta o resultado da renderização do exemplo, com um círculo verde preenchido à esquerda e um círculo apenas contornado à direita. Esse exemplo demonstra como a raylib oferece funções simples e diretas para a criação de formas geométricas básicas, facilitando a construção de interfaces e elementos gráficos em aplicações 2D.

Linhas

Código-fonte 14 Desenhando linhas simples e com espessura personalizada.

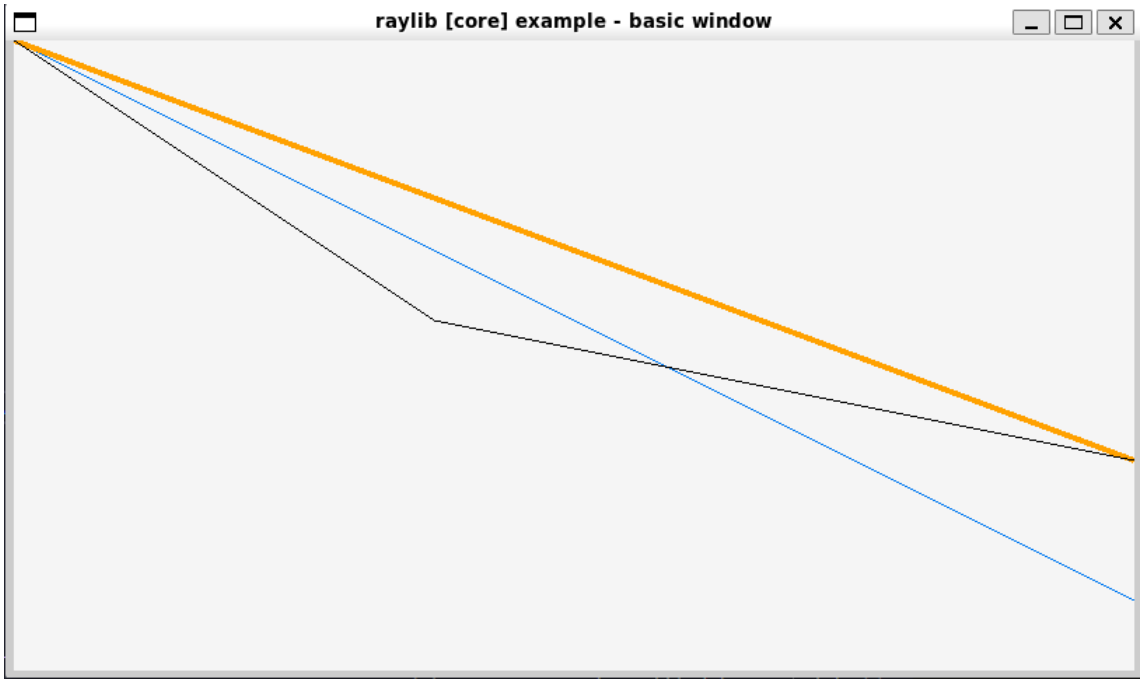
```
1 // DrawLine(inicioX, inicioY, fimX, fimY, cor)
2 DrawLine(0, 0, 800, 400, BLUE);
3
4 // Com espessura personalizada
5 DrawLineEx((Vector2){0, 0}, (Vector2){800, 300}, 4.0f, ORANGE);
6
7 // Com múltiplos pontos
8 DrawLineStrip((Vector2[]){(Vector2){0, 0}, (Vector2){300, 200},
↪  (Vector2){800, 300}}, 3, BLACK);
```

Fonte: Autoria própria.

O Código-fonte 14 demonstra diferentes formas de desenhar linhas utilizando a raylib. A função `DrawLine` permite traçar uma linha simples entre dois pontos definidos pelas coordenadas inicial e final. Já a função `DrawLineEx` oferece maior controle visual, permitindo especificar a espessura da linha, o que é útil para destacar elementos na interface. Por fim, a função `DrawLineStrip` conecta múltiplos pontos

em sequência, formando uma linha contínua composta por vários segmentos.

Figura 1.6: Renderização do algoritmo de exemplo 14.



Fonte: Autoria própria.

A Figura 1.6 ilustra o resultado da execução do Código-fonte, evidenciando os diferentes estilos de linhas: uma linha simples, uma linha com espessura maior e uma linha composta por múltiplos pontos conectados. Esse conjunto de funções demonstra a flexibilidade da raylib na criação de elementos gráficos básicos, essenciais para o desenvolvimento de interfaces e visualizações em aplicações 2D.

Cores

A Raylib inclui constantes de cor prontas para uso, como `RAYWHITE`, `BLACK`, `DARKGRAY`, `RED`, `GREEN`, `BLUE`, `YELLOW`, `ORANGE`, `PURPLE` e `WHITE`. Cores personalizadas são criadas com `(Color){R, G, B, A}`, onde cada canal vai de 0 a 255 e `A` representa a opacidade.

Código-fonte 15 Criando e usando uma cor personalizada RGBA.

```
1 Color minhaCor = (Color){ 100, 200, 150, 255 };
2 DrawRectangle(50, 50, 100, 100, minhaCor);
```

Fonte: Autoria própria.

O Código-fonte 15 exemplifica a criação de uma cor personalizada na raylib utilizando a estrutura `Color`, na qual são definidos os valores dos canais vermelho (R), verde (G), azul (B) e alfa (A). No exemplo, a cor `minhaCor` é composta por níveis intermediários de vermelho e azul, com maior intensidade de verde e opacidade total (`255`). Em seguida, essa cor é aplicada na função `DrawRectangle`, resultando no

desenho de um retângulo preenchido com a tonalidade especificada. Esse mecanismo permite maior controle sobre a paleta de cores utilizada na aplicação, indo além das constantes predefinidas pela biblioteca.

1.9 Entrada do usuário

Teclado

A Raylib oferece três funções principais para teclado:

Código-fonte 16 Funções de leitura de teclado: `IsKeyDown`, `IsKeyPressed` e `IsKeyReleased`.

```
// IsKeyDown     -- true ENQUANTO a tecla estiver pressionada
// IsKeyPressed  -- true apenas NO FRAME em que a tecla foi pressionada
// IsKeyReleased -- true apenas NO FRAME em que a tecla foi solta

if (IsKeyDown(KEY_SPACE))    { /* dispara continuamente */ }
if (IsKeyPressed(KEY_ENTER)) { /* confirma uma unica vez */ }
if (IsKeyReleased(KEY_UP))   { /* detecta fim do salto  */ }
```

Fonte: Autoria própria.

Constantes de teclado mais usadas: `KEY_RIGHT`, `KEY_LEFT`, `KEY_UP`, `KEY_DOWN`, `KEY_SPACE`, `KEY_ENTER`, `KEY_ESCAPE`, `KEY_W`, `KEY_A`, `KEY_S`, `KEY_D`.

Mouse

Código-fonte 17 Leitura de posição e botões do mouse.

```
// Posicao do cursor
int mx = GetMouseX();
int my = GetMouseY();
Vector2 mouse = GetMousePosition(); // como estrutura Vector2

// Botoes do mouse
if (IsMouseButtonDown(MOUSE_BUTTON_LEFT))    { /* arrastando  */ }
if (IsMouseButtonPressed(MOUSE_BUTTON_LEFT)) { /* clique unico */ }
```

Fonte: Autoria própria.

1.10 Compilando um projeto Raylib

Compilar com Raylib requer linkar as bibliotecas do sistema junto com o código. O processo varia conforme o sistema operacional.

Instalação da Raylib

Antes de compilar, a Raylib precisa estar disponível no sistema. A Tabela 1.3 lista os métodos recomendados para cada plataforma.

Tabela 1.3: Métodos de instalação da Raylib por plataforma.

Sistema	Comando
Ubuntu/Debian	`sudo apt install libraylib-dev`
Arch Linux	`sudo pacman -S raylib`
macOS (Homebrew)	`brew install raylib`
Windows (vcpkg)	`vcpkg install raylib`

Fonte: Autoria própria.

Também é possível baixar os binários pré-compilados diretamente em `https://github.com/raysan5/raylib/releases`.

Linux

Código-fonte 18 Comandos de compilação no Linux.

```
# Compilacao basica
gcc main.c -o app -lraylib -lm -lpthread -ldl -lrt -lX11

# Com warnings e otimizacao (recomendado durante o desenvolvimento)
gcc main.c -o app -Wall -O2 -lraylib -lm -lpthread -ldl -lrt -lX11

# Se a Raylib estiver em pasta local (nao instalada no sistema)
gcc main.c -o app -L./lib -I./include -lraylib -lm -lpthread -ldl -lrt
↪  -lX11
```

Fonte: Autoria própria.

A Tabela 1.4 descreve a função de cada flag utilizada no Linux.

Tabela 1.4: Flags de compilação no Linux e suas funções.

Flag	Função
`-lraylib`	Linka a biblioteca Raylib
`-lm`	Biblioteca matemática (`sin`, `cos`, `sqrt`…)
`-lpthread`	Suporte a threads POSIX
`-ldl`	Carregamento dinâmico de bibliotecas
`-lrt`	Funções de tempo real (`clock_gettime`)
`-lX11`	Sistema de janelas do Linux
`-Wall`	Ativa todos os avisos do compilador
`-O2`	Otimização de código para release
`-L./lib`	Pasta local de bibliotecas (`.so` / `.a`)
`-I./include`	Pasta local de cabeçalhos (`.h`)

Fonte: Autoria própria.

Windows (MinGW / GCC)

Como pode ser observado no Código-fonte 19, a compilação de programas desenvolvidos com a Raylib no Windows utilizando o compilador MinGW/GCC requer

a inclusão de algumas bibliotecas específicas do sistema operacional. Entre elas estão `-lopengl32`, responsável pelo acesso à API gráfica OpenGL; `-lgdi32`, utilizada para operações gráficas da interface GDI do Windows; e `-lwinmm`, que fornece suporte a recursos multimídia, como áudio e temporizadores. Além disso, a flag `-mwindows` pode ser utilizada para evitar a abertura de uma janela de terminal durante a execução da aplicação.

Código-fonte 19 Comandos de compilação no Windows com MinGW/GCC.

```
# Compilacao basica
gcc main.c -o app.exe -lraylib -lopengl32 -lgdi32 -lwinmm

# Com warnings e otimizacao
gcc main.c -o app.exe -Wall -O2 -lraylib -lopengl32 -lgdi32 -lwinmm

# Sem abrir janela de terminal junto ao jogo (modo GUI)
gcc main.c -o app.exe -mwindows -lraylib -lopengl32 -lgdi32 -lwinmm
```

Fonte: Autoria própria.

macOS

O Código-fonte 20 apresenta um exemplo de script para compilação de aplicações Raylib no macOS. O script reúne todos os comandos necessários para gerar o executável, facilitando o processo de construção do projeto. A utilização de scripts de compilação reduz a repetição de comandos, diminui a ocorrência de erros e torna o desenvolvimento mais organizado, especialmente em projetos maiores.

Código-fonte 20 Comandos de compilação no macOS.

```
# Com Homebrew (mais simples)
clang main.c -o app $(pkg-config --libs --cflags raylib)

# Manualmente, com os frameworks do sistema
clang main.c -o app -lraylib \
-framework OpenGL -framework Cocoa \
-framework IOKit -framework CoreAudio -framework CoreVideo
```

Fonte: Autoria própria.

Usando um Makefile

À medida que um projeto cresce e passa a utilizar múltiplos arquivos-fonte, a compilação manual pode se tornar trabalhosa. Nesses casos, a utilização de um *Makefile* simplifica significativamente o processo, automatizando a compilação e a organização das dependências entre os arquivos do projeto.

Código-fonte 21 Exemplo de Makefile para compilação de projetos Raylib.

```
CC     = gcc
CFLAGS = -Wall -O2
LIBS   = -lraylib -lm -lpthread -ldl -lrt -lX11

all: main.c
        $(CC) main.c -o app $(CFLAGS) $(LIBS)

clean:
        rm -f app
```

Fonte: Autoria própria.

Após criar o arquivo, basta executar o comando `make` para compilar o projeto. Para remover os arquivos gerados durante a compilação, utilize o comando `make clean`.

Dica: durante o desenvolvimento, recomenda-se compilar o projeto sem a flag de otimização `-O2`, pois isso reduz o tempo de compilação e facilita a depuração do código. A otimização pode ser adicionada posteriormente na geração da versão final da aplicação.

1.11 Exemplo prático: quadrado controlável

Esta seção desenvolve progressivamente uma pequena aplicação interativa, partindo do esqueleto básico e acrescentando, a cada etapa, um novo conceito: contenção nas bordas, coleta de pontos, sistema de vidas e tela de fim de jogo. Cada subseção apresenta apenas as partes novas ou modificadas em relação à anterior, seguidas do código completo da versão correspondente.

Versão 1 — movimento básico

O ponto de partida é um retângulo azul 40×40 pixels que se move pelo teclado (WASD ou setas direcionais) e muda de cor com os botões do mouse. O delta time garante velocidade constante independentemente do FPS.

O Código-fonte 22 apresenta a definição de uma estrutura chamada `Player`, utilizada para representar um objeto no jogo. Essa estrutura agrupa informações essenciais como a posição (`x` e `y`), a velocidade de movimento (`speed`) e a cor (`color`). A utilização de `struct` permite organizar melhor os dados relacionados a uma entidade, facilitando a manipulação e expansão do código, especialmente em projetos que envolvem múltiplos objetos com características semelhantes.

Código-fonte 22 Estrutura do Player.

```
#include "raylib.h"

typedef struct{
    float x;
    float y;
    float speed;
    int size;
    Color color;
} Player;
```

Fonte: Autoria própria.

O Código-fonte 23 apresenta a etapa de renderização da aplicação utilizando a raylib. A função `BeginDrawing` inicia o processo de desenho na tela, enquanto `ClearBackground` define a cor de fundo para o frame atual. Em seguida, a função `DrawRectangle` é utilizada para desenhar o jogador na posição atual, convertendo suas coordenadas para inteiros e aplicando a cor definida na estrutura `p1`. Além disso, `DrawText` exibe instruções na tela para orientar o usuário quanto aos controles de movimentação e interação. Por fim, a função `DrawFPS` mostra a taxa de quadros por segundo em tempo real, sendo útil para monitoramento de desempenho durante a execução.

Código-fonte 23 Código para renderização da tela.

```
        ClearBackground(RAYWHITE);
        DrawRectangle((int)p1.x, (int)p1.y, 40, 40, p1.color);
        DrawText("WASD ou setas para mover", 10, 10, 18, DARKGRAY);
        DrawText("Clique para mudar a cor",  10, 34, 18, DARKGRAY);
        DrawFPS(700, 10);
        EndDrawing();
```

Fonte: Autoria própria.

Como pode ser visto no Código-fonte 24, foi criado uma estrutura para o jogador, contendo sua posição (x,y), sua velocidade, tamanho e cor.

Código-fonte 24 Versão 1: quadrado controlável com teclado e mouse (Exemplo Completo).

```
1  #include "raylib.h"
2
3  typedef struct{
4      float x;
5      float y;
6      float speed;
7      int size;
8      Color color;
9  } Player;
```

Fonte: Autoria própria.

No Código-fonte 25 é apresentada a função responsável pela leitura das entradas do teclado e pelo controle da movimentação do jogador. A função verifica continuamente se determinadas teclas estão pressionadas utilizando a função `IsKeyDown()`, permitindo a movimentação do personagem em quatro direções.

As teclas direcionais e as teclas `W`, `A`, `S` e `D` são utilizadas como alternativas para o controle do movimento. Quando a tecla correspondente à direita é pressionada, a coordenada `x` do jogador é incrementada, deslocando o personagem horizontalmente para a direita. De forma semelhante, ao pressionar a tecla da esquerda, a coordenada `x` é decrementada, realizando o movimento para a esquerda.

Para o movimento vertical, ao pressionar a tecla para baixo, a coordenada `y` é incrementada, enquanto a tecla para cima decrementa essa coordenada, movendo o personagem para cima da tela.

A velocidade do movimento é calculada utilizando o atributo `speed` multiplicado pela variável `dt` (*delta time*), garantindo que a movimentação do jogador permaneça consistente independentemente da taxa de quadros por segundo do jogo.

Código-fonte 25 Função de leitura do teclado e controle do jogador.

```
11 void leitura_teclado(Player *p1){
12     if (IsKeyDown(KEY_RIGHT) || IsKeyDown(KEY_D))
13             p1.x += p1.speed * dt;
14         if (IsKeyDown(KEY_LEFT)  || IsKeyDown(KEY_A))
15             p1.x -= p1.speed * dt;
16         if (IsKeyDown(KEY_DOWN)  || IsKeyDown(KEY_S))
17             p1.y += p1.speed * dt;
18         if (IsKeyDown(KEY_UP)    || IsKeyDown(KEY_W))
19             p1.y -= p1.speed * dt;
20 }
```

Fonte: Autoria própria.

No Código-fonte 26 temos o código completo do exemplo. Mostrando todas

as partes que foram contruidas anteriormente.

Código-fonte 26 Versão 1: quadrado controlável com teclado e mouse (Exemplo Completo).

```
22 int main(void) {
23     InitWindow(800, 600, "Quadrado Controlavel");
24     SetTargetFPS(60);
25
26     Player p1;
27     p1.x = 380.0f;
28     p1.y = 280.0f;
29     p1.speed = 250.0f;
30     p1.color = BLUE;
31
32     while (!WindowShouldClose()) {
33         float dt = GetFrameTime();
34
35         leitura_teclado(&p1);
36
37         // Muda de cor ao clicar com o mouse
38         if (IsMouseButtonPressed(MOUSE_BUTTON_LEFT))
39             p1.color = RED;
40
41         if (IsMouseButtonPressed(MOUSE_BUTTON_RIGHT))
42             p1.color = BLUE;
43
44         BeginDrawing();
45         ClearBackground(RAYWHITE);
46         DrawRectangle((int)p1.x, (int)p1.y, 40, 40, p1.color);
47         DrawText("WASD ou setas para mover", 10, 10, 18, DARKGRAY);
48         DrawText("Clique para mudar a cor",  10, 34, 18, DARKGRAY);
49         DrawFPS(700, 10);
50         EndDrawing();
51     }
52
53     CloseWindow();
54     return 0;
55 }
```

Fonte: Autoria própria.

Versão 2 — contenção nas bordas

Sem restrição de movimento, o quadrado pode sair completamente da janela. A solução é *clampar* as coordenadas após cada atualização, garantindo que o jogador permaneça sempre visível.

O que muda: após o bloco de entrada, adicione as duas linhas de *clamp* abaixo. A váriavel `size` da struct `Player` centraliza o tamanho do quadrado para facilitar ajustes futuros.

Código-fonte 27 Estrutura da Janela.

```
typedef struct
{
    int width;
    int height;
} Screen;
```

Fonte: Autoria própria.

O Código-fonte 27 define a estrutura `Screen`, utilizada para armazenar as dimensões da janela, por meio dos campos `width` e `height`. Essa estrutura é importante para centralizar as informações de tamanho da tela, facilitando a organização do código e permitindo implementar verificações de colisão que impedem o jogador de ultrapassar os limites visíveis da aplicação.

Código-fonte 28 código para definição dos limites da tela.

```
        // Contem o quadrado dentro da janela
        if (p1.x < 0)
            p1.x = 0;

        if (p1.x > tela.width - p1.size)
            p1.x = tela.width - p1.size;

        if (p1.y < 0)
            p1.y = 0;

        if (p1.y > tela.height - p1.size)
            p1.y = tela.height - p1.size;
```

Fonte: Autoria própria.

O Código-fonte 28 implementa a lógica de limitação de movimento do jogador, garantindo que ele permaneça dentro dos limites da tela. Para isso, são realizadas verificações nas coordenadas `x` e `y` do objeto `p1`. Caso o jogador ultrapasse os limites mínimos (0) ou máximos (definidos pelas dimensões da tela menos o tamanho do próprio objeto), sua posição é ajustada para o valor permitido mais próximo. Esse processo, conhecido como *clamp*, é fundamental para evitar que o jogador saia da área visível da aplicação.

Código-fonte 29 Função de Controle de pontuação e redefinição de posição dos alvos.

```
1  void pontuar(Player player, Circle *alvo, Screen tela, int *pontos)
2  {
3      if (CheckCollisionCircleRec((Vector2){(*alvo).x, (*alvo).y},
   ↪  (*alvo).radius, (Rectangle){player.x, player.y, player.size,
   ↪  player.size}))
4      {
5          (*pontos)++;
6          (*alvo).x = (float)GetRandomValue((*alvo).radius, tela.width -
   ↪  (*alvo).radius);
7          (*alvo).y = (float)GetRandomValue((*alvo).radius, tela.height -
   ↪  (*alvo).radius);
8      }
9  }
```

Fonte: Autoria própria.

O trecho apresentado no Código-fonte 30 define as estruturas de dados utilizadas no jogo: `Player`, `Screen` e `Circle`.

Código-fonte 30 Definição das estruturas de dados.

```
2  #include "raylib.h"
3
4  typedef struct
5  {
6      float x;
7      float y;
8      float speed;
9      int size;
10     Color color;
11 } Player;
12
13 typedef struct
14 {
15     int width;
16     int height;
17 } Screen;
18
19 typedef struct
20 {
21     float x;
22     float y;
23     float radius;
24 } Circle;
```

Fonte: Autoria própria.

A função `pontuar` (Código-fonte 29) verifica colisão entre o jogador e o alvo, incrementa a pontuação e reposiciona o alvo aleatoriamente.

Código-fonte 31 Função `pontuar`: detecção de colisão e atualização do alvo.

```
26 void pontuar(Player player, Circle *alvo, Screen tela, int *pontos)
27 {
28     if (CheckCollisionCircleRec((Vector2){(*alvo).x, (*alvo).y},
   ↪  (*alvo).radius, (Rectangle){player.x, player.y, player.size,
   ↪  player.size}))
29     {
30         (*pontos)++;
31         (*alvo).x = (float)GetRandomValue((*alvo).radius, tela.width -
   ↪  (*alvo).radius);
32         (*alvo).y = (float)GetRandomValue((*alvo).radius, tela.height -
   ↪  (*alvo).radius);
33     }
34 }
```

Fonte: Autoria própria.

A função `mover_jogador` presente no Código-fonte 32 lê as teclas pressionadas e atualiza a posição do jogador com base no *delta time*.

Código-fonte 32 Função `mover_jogador`: movimentação via teclado.

```
36 void mover_jogador(Player *p1, float dt)
37 {
38     if (IsKeyDown(KEY_RIGHT) || IsKeyDown(KEY_D))
39         p1->x += p1->speed * dt;
40     if (IsKeyDown(KEY_LEFT) || IsKeyDown(KEY_A))
41         p1->x -= p1->speed * dt;
42     if (IsKeyDown(KEY_DOWN) || IsKeyDown(KEY_S))
43         p1->y += p1->speed * dt;
44     if (IsKeyDown(KEY_UP) || IsKeyDown(KEY_W))
45         p1->y -= p1->speed * dt;
46 }
```

Fonte: Autoria própria.

A função `manter_na_tela` que pode ser vista no Código-fonte 33) impede que o jogador ultrapasse os limites da janela, corrigindo sua posição quando necessário.

Código-fonte 33 Função `manter_na_tela`: restrição do jogador aos limites da janela.

```
48 void manter_na_tela(Player *p1, Screen tela)
49 {
50     if (p1->x < 0)
51         p1->x = 0;
52
53     if (p1->x > tela.width - p1->size)
54         p1->x = tela.width - p1->size;
55
56     if (p1->y < 0)
57         p1->y = 0;
58
59     if (p1->y > tela.height - p1->size)
60         p1->y = tela.height - p1->size;
61 }
```

Fonte: Autoria própria.

A função `atualizar_cor` do Código-fonte 34 altera a cor do jogador conforme o botão do mouse pressionado.

Código-fonte 34 Função `atualizar_cor`: alteração de cor com o mouse.

```
63 void atualizar_cor(Player *p1)
64 {
65     if (IsMouseButtonPressed(MOUSE_BUTTON_LEFT))
66         p1->color = RED;
67     if (IsMouseButtonPressed(MOUSE_BUTTON_RIGHT))
68         p1->color = BLUE;
69 }
```

Fonte: Autoria própria.

A função `desenhar_jogo` que pode ser vista no Código-fonte 35 agrupa todas as chamadas de renderização, desenhando o jogador, o alvo, os textos e o FPS na tela.

Código-fonte 35 Função `desenhar_jogo`: renderização dos elementos na tela.

```
71 void desenhar_jogo(Player p1, Circle alvo, Screen tela, int pontos)
72 {
73     BeginDrawing();
74     ClearBackground(RAYWHITE);
75     DrawRectangle((int)p1.x, (int)p1.y, p1.size, p1.size, p1.color);
76     DrawCircle(alvo.x, alvo.y, alvo.radius, GREEN);
77     DrawText("WASD ou setas para mover \nCapture os pontos verdes", 10,
   ↪  10, 18, DARKGRAY);
78     DrawText(TextFormat("Pontos: %d", pontos), 10, tela.height - 30, 30,
   ↪  BLACK);
79     DrawFPS(700, 10);
80     EndDrawing();
81 }
```

Fonte: Autoria própria.

A função `main` que foi representada no Código-fonte 36 inicializa as variáveis, abre a janela e executa o loop principal do jogo, chamando cada função na ordem correta.

Código-fonte 36 Função `main`: inicialização e loop principal.

```
83  int main(void)
84  {
85      Screen tela = {800, 600};
86      Circle alvo = {150.0f, 300.0f, 20.0f};
87      Player p1;
88      p1.x = 380.0f;
89      p1.y = 280.0f;
90      p1.speed = 250.0f;
91      p1.size = 40.0f;
92      p1.color = BLUE;
93
94      int pontos = 0;
95
96      InitWindow(tela.width, tela.height, "Quadrado Controlavel v2");
97      SetTargetFPS(60);
98
99      while (!WindowShouldClose())
100     {
101         float dt = GetFrameTime();
102
103         mover_jogador(&p1, dt);
104         manter_na_tela(&p1, tela);
105         atualizar_cor(&p1);
106
107         pontuar(p1, &alvo, tela, &pontos);
108
109         desenhar_jogo(p1, alvo, tela, pontos);
110     }
111
112     CloseWindow();
113     return 0;
114 }
```

Fonte: Autoria própria.

Versão 3 — alvos e pontuação

Introduzimos um *alvo* circular que aparece em posição aleatória. Quando o jogador toca o alvo, a pontuação aumenta e o alvo reaparece em outro lugar. Usamos `CheckCollisionCircleRec` para detectar a sobreposição entre o quadrado e o círculo, e `GetRandomValue` para sortear a nova posição. **Novidades em relação à versão anterior:** inimigo controlado por IA que persegue o jogador; sistema de vidas com `checar_colisao_inimigo`; tela de *game over* com opção de reinício via tecla `R`.

As estruturas de dados do Código-fonte 37 definem os tipos `Player`, `Screen` e `Circle`, agora com o campo `vidas` em `Player`.

Código-fonte 37 Definição das estruturas de dados.

```
#include "raylib.h"

typedef struct
{
    float x;
    float y;
    float speed;
    int size;
    int vidas;
    Color color;
} Player;

typedef struct
{
    int width;
    int height;
} Screen;

typedef struct
{
    float x;
    float y;
    float radius;
} Circle;
```

Fonte: Autoria própria.

A função `pontuar`, apresentada no Código-fonte 38, é responsável por verificar a colisão entre o jogador e o alvo circular. Quando a colisão é detectada, a pontuação do jogador é incrementada e uma nova posição aleatória é gerada para o alvo, permitindo que a partida continue. Essa abordagem cria um ciclo contínuo de interação, no qual o jogador deve alcançar repetidamente o alvo para aumentar sua pontuação.

Código-fonte 38 Função `pontuar`: colisão com o alvo e atualização da pontuação.

```
26 void pontuar(Player player, Circle *alvo, Screen tela, int *pontos)
27 {
28     if (CheckCollisionCircleRec((Vector2){(*alvo).x, (*alvo).y},
   ↪  (*alvo).radius, (Rectangle){player.x, player.y, player.size,
   ↪  player.size}))
29     {
30         (*pontos)++;
31         (*alvo).x = (float)GetRandomValue((*alvo).radius, tela.width -
   ↪  (*alvo).radius);
32         (*alvo).y = (float)GetRandomValue((*alvo).radius, tela.height -
   ↪  (*alvo).radius);
33     }
34 }
```

Fonte: Autoria própria.

A função `move_inimigo` do Código-fonte 39 move o inimigo em direção ao jogador a cada quadro, aproximando-se nos eixos x e y com base na velocidade e no *delta time*.

Código-fonte 39 Função `move_inimigo`: perseguição do jogador pela IA.

```
36 void move_inimigo(Player *inimigo, Player player, float dt)
37 {
38     if (player.x > (*inimigo).x)
39         (*inimigo).x += (*inimigo).speed * dt;
40     else if (player.x < (*inimigo).x)
41         (*inimigo).x -= (*inimigo).speed * dt;
42
43     if (player.y > (*inimigo).y)
44         (*inimigo).y += (*inimigo).speed * dt;
45     else if (player.y < (*inimigo).y)
46         (*inimigo).y -= (*inimigo).speed * dt;
47 }
```

Fonte: Autoria própria.

A função `mover_jogador` (código 40) lê as teclas pressionadas e atualiza a posição do jogador com base no *delta time*.

Código-fonte 40 Função `mover_jogador`: movimentação via teclado.

```
49 void mover_jogador(Player *p1, float dt)
50 {
51     if (IsKeyDown(KEY_RIGHT) || IsKeyDown(KEY_D))
52         p1->x += p1->speed * dt;
53     if (IsKeyDown(KEY_LEFT) || IsKeyDown(KEY_A))
54         p1->x -= p1->speed * dt;
55     if (IsKeyDown(KEY_DOWN) || IsKeyDown(KEY_S))
56         p1->y += p1->speed * dt;
57     if (IsKeyDown(KEY_UP) || IsKeyDown(KEY_W))
58         p1->y -= p1->speed * dt;
59 }
```

Fonte: Autoria própria.

A função `manter_na_tela` (código 41) impede que o jogador ultrapasse os limites da janela, corrigindo sua posição quando necessário.

Código-fonte 41 Função `manter_na_tela`: restrição do jogador aos limites da janela.

```
61 void manter_na_tela(Player *p1, Screen tela)
62 {
63     if (p1->x < 0)
64         p1->x = 0;
65
66     if (p1->x > tela.width - p1->size)
67         p1->x = tela.width - p1->size;
68
69     if (p1->y < 0)
70         p1->y = 0;
71
72     if (p1->y > tela.height - p1->size)
73         p1->y = tela.height - p1->size;
74 }
```

Fonte: Autoria própria.

A função `atualizar_cor` (código 1) altera a cor do jogador conforme o botão do mouse pressionado.

Listing 1 Função `atualizar_cor`: alteração de cor com o mouse.

```
76 void atualizar_cor(Player *p1)
77 {
78     if (IsMouseButtonPressed(MOUSE_BUTTON_LEFT))
79         p1->color = RED;
80     if (IsMouseButtonPressed(MOUSE_BUTTON_RIGHT))
81         p1->color = BLUE;
82 }
```

Fonte: Autoria própria.

A função `checar_colisao_inimigo` (código 2) verifica se o jogador colidiu com o inimigo, decrementa uma vida e reposiciona o inimigo aleatoriamente; caso as vidas cheguem a zero, o estado do jogo é alterado para *game over*.

Listing 2 Função `checar_colisao_inimigo`: colisão com inimigo e controle de vidas.

```
84 void checar_colisao_inimigo(Player *p1, Player *inimigo, Screen tela, int
   ↪ *estado)
85 {
86     if (p1->vidas <= 0)
87     {
88         *estado = 0;
89     }
90     else if (CheckCollisionRecs((Rectangle){p1->x, p1->y, p1->size,
   ↪ p1->size}, (Rectangle){inimigo->x, inimigo->y, inimigo->size,
   ↪ inimigo->size}))
91     {
92         p1->vidas--;
93         inimigo->x = (float)GetRandomValue(0, tela.width -
   ↪ inimigo->size);
94         inimigo->y = (float)GetRandomValue(0, tela.height -
   ↪ inimigo->size);
95     }
96 }
```

Fonte: Autoria própria.

A função `desenhar_jogo` (código 3) agrupa todas as chamadas de renderização, desenhando o jogador, o inimigo, o alvo, o placar de pontos, as vidas restantes e o FPS.

Listing 3 Função `desenhar_jogo`: renderização dos elementos na tela.

```
98  void desenhar_jogo(Player p1, Player inimigo, Circle alvo, Screen tela,
    ↪  int pontos)
99  {
100     BeginDrawing();
101     ClearBackground(RAYWHITE);
102     DrawRectangle((int)p1.x, (int)p1.y, p1.size, p1.size, p1.color);
103     DrawRectangle((int)inimigo.x, (int)inimigo.y, inimigo.size,
    ↪  inimigo.size, inimigo.color);
104     DrawCircle(alvo.x, alvo.y, alvo.radius, GREEN);
105     DrawText("WASD ou setas para mover \nCapture os pontos verdes", 10,
    ↪  10, 18, DARKGRAY);
106     DrawText(TextFormat("Vidas: %d", p1.vidas), 10, tela.height - 70, 30,
    ↪  BLACK);
107     DrawText(TextFormat("Pontos: %d", pontos), 10, tela.height - 30, 30,
    ↪  BLACK);
108     DrawFPS(700, 10);
109     EndDrawing();
110 }
```

Fonte: Autoria própria.

A função `reiniciar_jogo` presente no Código-fonte 42 aguarda o pressionamento da tecla `R` para restaurar o estado do jogo, zerando a pontuação, as vidas e as posições do jogador e do inimigo.

Código-fonte 42 Função `reiniciar_jogo`: reset do estado ao pressionar `R`.

```
112 void reiniciar_jogo(int *estado, int *pontos, Player *p1, Player
    ↪  *inimigo)
113 {
114     if(IsKeyPressed(KEY_R))
115     {
116         *estado = 1;
117         *pontos = 0;
118         p1->vidas = 3;
119         p1->x = 380.0f;
120         p1->y = 280.0f;
121         inimigo->x = 100.0f;
122         inimigo->y = 100.0f;
123     }
124 }
```

Fonte: Autoria própria.

A função `desenhar_game_over` do Código-fonte 43 exibe a tela de *game over* com a pontuação final e a instrução para reiniciar.

Código-fonte 43 Função `desenhar_game_over`: tela de encerramento da partida.

```
126 void desenhar_game_over(Screen tela, int pontos)
127 {
128     BeginDrawing();
129     ClearBackground(BLACK);
130 
131     DrawText("GAME OVER",
132              tela.width / 2 - MeasureText("GAME OVER", 60) / 2,
133              180, 60, RED);
134     DrawText(TextFormat("Pontuacao final: %d", pontos),
135              tela.width / 2 - MeasureText("Pontuacao final: 00", 32) / 2,
136              280, 32, RAYWHITE);
137     DrawText("Pressione R para jogar novamente",
138              tela.width / 2 -
139                  MeasureText("Pressione R para jogar novamente", 20) / 2,
140              360, 20, GRAY);
141     EndDrawing();
142 }
```

Fonte: Autoria própria.

A função `main`, apresentada no Código-fonte 44, é o ponto de entrada da aplicação. Nela são inicializadas as estruturas que representam a tela, o jogador, o inimigo e o alvo, além das variáveis responsáveis pelo controle do estado do jogo e da pontuação.

Após a criação da janela e a definição da taxa de atualização por meio da função `SetTargetFPS`, inicia-se o laço principal da aplicação. Esse laço é executado continuamente enquanto a janela permanecer aberta, sendo responsável por atualizar a lógica do jogo e desenhar os elementos na tela a cada quadro.

Durante a execução, o programa verifica o estado atual do jogo. Quando o jogador perde todas as vidas, o estado é alterado para *game over*, fazendo com que seja exibida uma tela de encerramento. Nesse modo, o usuário pode reiniciar a partida pressionando a tecla `R`. Caso o jogo esteja ativo, são executadas as funções de movimentação do jogador, atualização da cor do personagem, detecção de colisões, movimentação do inimigo, contabilização da pontuação e renderização dos elementos gráficos.

Essa organização centraliza o fluxo de execução da aplicação e demonstra um padrão amplamente utilizado no desenvolvimento de jogos, conhecido como *game loop*, no qual as etapas de entrada, atualização e desenho são repetidas continuamente até o encerramento do programa.

Código-fonte 44 Função `main`: inicialização e loop principal do jogo.

```
144 int main(void)
145 {
146     Screen tela = {800, 600};
147     Circle alvo = {150.0f, 300.0f, 20.0f};
148     Player p1;
149     p1.x = 380.0f;
150     p1.y = 280.0f;
151     p1.speed = 250.0f;
152     p1.size = 40.0f;
153     p1.color = BLUE;
154     p1.vidas = 3;
155 
156     Player inimigo = {100.0f, 100.0f, 100.0f, 40.0f, 0, RED};
157 
158     int estado = 1; // 1: rodando, 0: gameover
159     int pontos = 0;
160 
161     InitWindow(tela.width, tela.height, "Quadrado Controlavel v2");
162     SetTargetFPS(60);
163 
164     while (!WindowShouldClose())
165     {
166         if (estado == 0)
167         {
168             reiniciar_jogo(&estado, &pontos, &p1, &inimigo);
169             desenhar_game_over(tela, pontos);
170             continue;
171         }
172 
173         float dt = GetFrameTime();
174 
175         mover_jogador(&p1, dt);
176         manter_na_tela(&p1, tela);
177         atualizar_cor(&p1);
178 
179         checar_colisao_inimigo(&p1, &inimigo, tela, &estado);
180 
181         pontuar(p1, &alvo, tela, &pontos);
182         move_inimigo(&inimigo, p1, dt);
183 
184         desenhar_jogo(p1, inimigo, alvo, tela, pontos);
185     }
186 
187     CloseWindow();
188     return 0;
189 }
```

Fonte: Autoria própria.

Tabela 1.5: Evolução das versões do exemplo prático.

Versão	Título	Conceitos introduzidos
1	Movimento básico	Loop, delta time, teclado, mouse
2	Contenção nas bordas	Clamp de coordenadas, constantes de tamanho
3	Alvos e pontuação	`GetRandomValue`, `CheckCollisionRecs`, `TextFormat`
4	Jogo completo	Vidas, obstáculos, dificuldade crescente, máquina de estados, reinício sem fechar janela

Fonte: Autoria própria.

A Tabela 1.5 apresenta uma visão progressiva do desenvolvimento do projeto, evidenciando como novas funcionalidades e conceitos foram incorporados a cada versão. Parte-se de uma implementação inicial simples, focada no controle básico de movimento, e evolui-se gradualmente para um sistema mais completo, com mecânicas de jogo, gerenciamento de estados e aumento dinâmico da dificuldade. Essa organização incremental facilita a compreensão do processo de construção do jogo, além de destacar a introdução estruturada de conceitos fundamentais ao longo das etapas.

1.12 Considerações finais

Este minicurso apresentou os fundamentos necessários para criar aplicações gráficas em C com a biblioteca Raylib: a estrutura básica de programa, o funcionamento do loop de janela, o controle de FPS com `SetTargetFPS`, as principais funções de desenho, a leitura de teclado e mouse, e o processo de compilação multiplataforma.

A partir desses conceitos, é possível evoluir para tópicos mais avançados como carregamento de imagens e texturas, reprodução de áudio, colisão entre formas, câmeras 2D e até renderização 3D — todos disponíveis na própria Raylib. A documentação completa e centenas de exemplos estão disponíveis em `https://raylib.com` e no repositório oficial `https://github.com/raysan5/raylib`.

1.13 Declaração de uso de IA generativa

☐ NÃO utilizei ferramentas de IA generativa para produzir, reescrever ou expandir trechos do texto do meu mini-curso. Eventuais correções automáticas de ortografia/gramática do editor de texto não se enquadram nesta declaração.

☑ UTILIZEI ferramentas de IA generativa de forma ética e transparente, exclusivamente como apoio redacional (ex.: revisão gramatical, melhoria de clareza, sugestão de organização), sem terceirização do conteúdo intelectual. Declaro que: (a) informei o uso; (b) revisei integralmente e sou responsável(a) pelo texto final; (c) conferi a veracidade de dados, citações e referências, garan-

tindo que não há referências inventadas, trechos sem fonte ou conteúdo não verificável.

Se marcou acima a opção UTILIZEI, descreva de forma objetiva:

- **Ferramenta(s)/versão(ões):** GPT-5.5 Free e Claude Sonnet 4.6 free;
- **Seções/trechos impactados:** Todo o texto;
- **Finalidade(s):** Geração de pequenos trechos de texto e revisão ortográfica.

1.14 Bibliografia

Albuquerque, E., Passos, J. V., Ferreira, M., Bezerra, P., and Barros, T. (2025). Formula c: Velocidade em um novo ângulo. In *Simpósio Brasileiro de Jogos e Entretenimento Digital (SBGames)*, pages 411–416. SBC.

Linux Foundation (2026). Linux. https://www.linux.org. Acesso em: 11 jun. 2026.

Microsoft (2026). Microsoft windows. https://www.microsoft.com/windows. Acesso em: 11 jun. 2026.

Raylib Technologies (2026). raylib: a simple and easy-to-use library to enjoy videogames programming. https://www.raylib.com. Acesso em: 10 jun. 2026.

Visual Studio Code (2026). Visual studio code. https://code.visualstudio.com/. Acessado em 16 maio 2026.

Minicurso 2

Prática com Docker e Docker Compose

Guilherme Carlos Freitas Bartasson e Eduardo Augusto Duarte de Castro Giovannetti

Resumo

Este artigo apresenta um minicurso introdutório sobre o uso de Docker e Docker Compose para criação, gerenciamento e orquestração de containers em ambientes virtualizados. O trabalho aborda desde a configuração inicial de uma máquina virtual Ubuntu e instalação do Docker até a construção de imagens, execução de containers, persistência de dados com volumes e integração de serviços utilizando Docker Compose. Além disso, são explorados conceitos fundamentais como Dockerfile, redes Docker e automação de ambientes, utilizando aplicações práticas desenvolvidas em Node.js e banco de dados MySQL. Como complemento, o minicurso também demonstra o uso da ferramenta Portainer para gerenciamento visual de containers, proporcionando aos participantes uma visão prática e aplicada das tecnologias amplamente utilizadas em ambientes corporativos modernos.

2.1 Introdução

Neste minicurso introdutório, a ideia principal é instalar e configurar um ambiente virtual, com algumas aplicações interagindo entre si, usando Docker e Docker Compose para criar containers de aplicações.

No processo iremos abordar: o que é o Docker, algumas características e propriedades do Docker, como instalar e configurar o Docker, como funcionam as aplicações que utilizam essa ferramenta e sua importância para os ambientes corporativos atuais. Serão abordadas também algumas ferramentas essenciais para gerenciar e manter este tipo de ambiente.

2.2 Público-alvo

O público-alvo do minicurso são pessoas que possuem interesse no desenvolvimento web e de aplicações de forma geral. O curso requer familiaridade iniciante com Linux e o seu terminal, e familiaridade iniciante com a virtualização de máquinas.

2.3 Pré-requisitos do curso

O minicurso utilizará uma máquina virtual (VM) Ubuntu LTS 22.04.5 feita através do VirtualBox 7.0.16, para executar o Docker CE 29.2.1.

2.4 Pré-requisitos de usuário

Para replicar o curso, o usuário precisará utilizar um sistema que permita virtualização e a instalação dos seguintes requisitos:

- Biblioteca Node.js v24.14.0 ou superior;
- Biblioteca NPM 11.9.0 ou superior;
- Git 2.53.0 ou superior;
- Visual Studio Code 1.109 ou superior;
- OpenSSH 8.9p1 ou superior;
- bash-completion 2.17.0 ou superior;

2.5 Fundamentos Teóricos

Apresentação Teórica

Vamos iniciar a aula com uma apresentação em slideshow teórica, entregando uma visão geral sobre o conteúdo do minicurso, baseando-se em GOMES [2016], incluindo o que é o Docker, sua importância na arquitetura digital moderna, o que é um container, o que é uma imagem de sistema, o que é o DockerFile, o que é o Docker Compose, e como é estruturado um arquivo Compose.

2.6 Prática com Docker

Configuração do ambiente e instalação do Docker

Este primeiro momento será dedicado à configuração do nosso ambiente de trabalho, para isso vamos inicializar as máquinas virtuais e montar a conexão via SSH

usando o VsCode, e também vamos ensinar a instalar o Docker no servidor com seus autocompletes padrões.

Configurações da Máquina Virtual e ambiente SSH no VScode

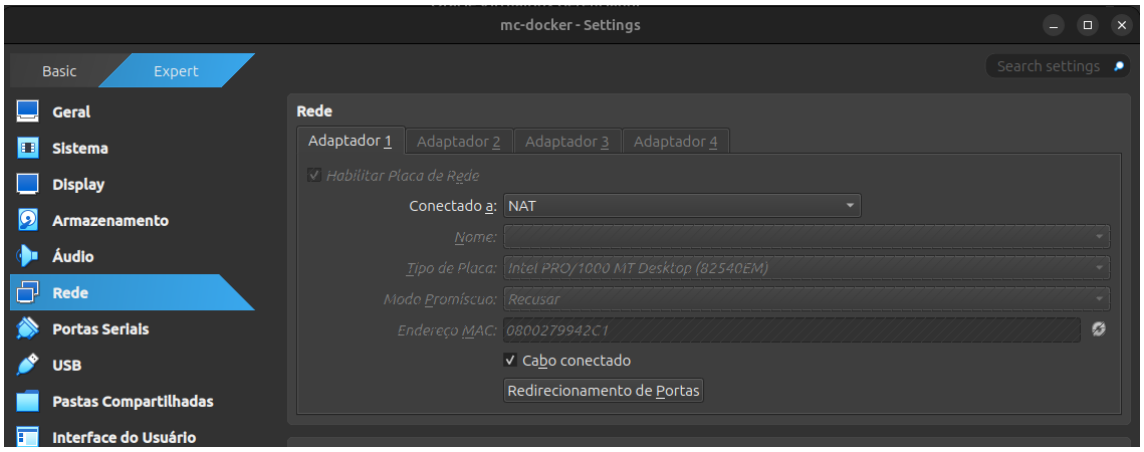
Para a máquina virtual, estaremos disponibilizando em um arquivo OVA (Open Virtual Appliance) um ambiente Ubuntu 22.04.5 já pré-configurado com todos os arquivos bases, pacotes, todas dependências necessárias já instaladas na máquina e usuário: **aluno** e senha: **curso** já configurado, cabendo ao aluno apenas a seguir os próximos passos corretamente para configurar o ambiente em sua máquina. Primeiramente, é necessário garantir que a rede da VM esteja configurada corretamente, para isso em seu VirtualBox acesse as configurações de Rede da VM e garanta que ela esteja em modo **NAT** e logo abaixo em **Avançado; Redirecionamento de Porta**, o endereço **localhost** na porta **2222** esteja configurado com a saida **22** para o servidor, de modo que o localhost (127.0.0.1) na porta 2222 seja redirecionado para a porta 22 do SSH do servidor, assim conseguiremos conectar via ssh no servidor, veja as Figuras 2.1 e 2.2. Após isso, é se torna possível a conexão SSH através do comando:

Código-fonte 45 Comando padrão para conexão ssh usando a porta 2222.

```
1 $ ssh 127.0.0.1@aluno -p 2222
```

Fonte: Autoria própria.

Figura 2.1: Configuração de rede no VirtualBox.



Fonte: Autoria própria.

Figura 2.2: Redirecionamento de portas configurado no VirtualBox.

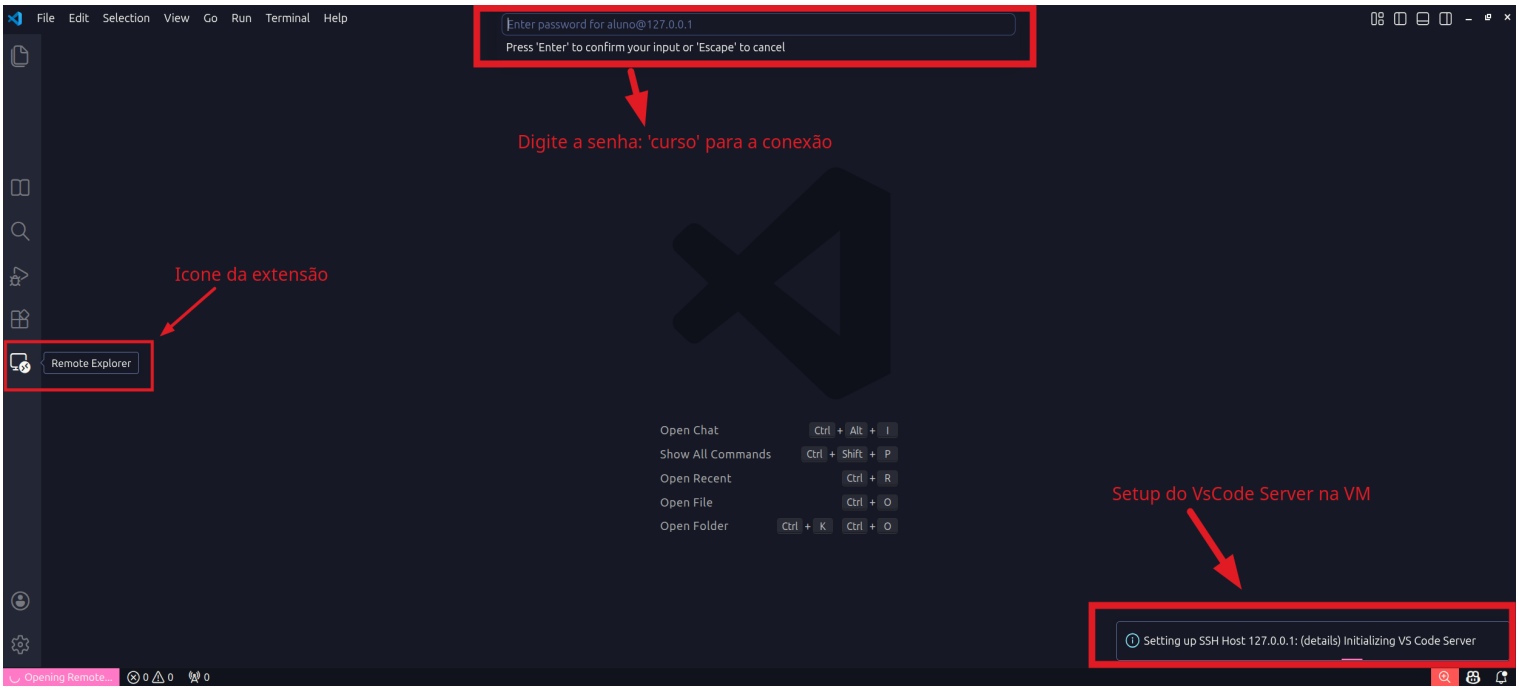
Fonte: Autoria própria.

Feita essa primeira conexão, esse mesmo host será adicionado no histórico SSH da sua máquina oque nos permite a conexão SSH com ele através do VsCode usando a extensão Remote - SSH da microsoft. Para isso abra seu **VsCode - Extensions** e procure por **Remote - SSH** veja a Figura 2.3, após a instalação dela, basta selecionar a extensão que uma lista com todos os host disponíveis irá aparecer, agora basta se conectar no servidor selecionando o mesmo host que usamos para o comando acima, que já estará no histórico de conexão e no processo selecionar a pasta *minicurso* como diretório em **Open Folder**. Vejas as Figuras 2.4 e 2.5, onde uma mostra a tela de conexão com o host e a outra a seleção de pasta dentro do host junto do terminal da VM já conectado.

Figura 2.3: Extensão Remote - SSH no VSCode.

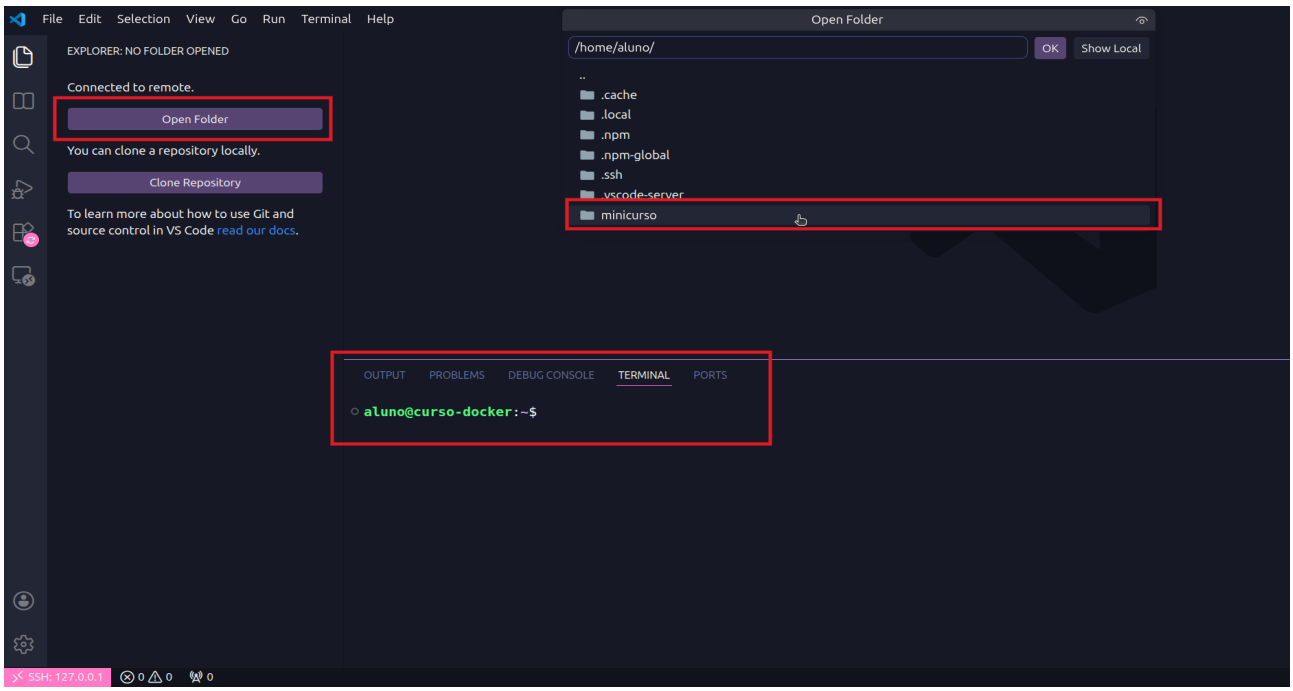
Fonte: Autoria própria.

Figura 2.4: Tela de conexão com host + icone da extensão no VsCode.



Fonte: Autoria própria.

Figura 2.5: Seleção da pasta Minicurso no host + Terminal conectado.



Fonte: Autoria própria.

Instalação do Docker

Na interface do VScode vamos ter acesso ao nosso diretório principal */minicurso*, onde estão todos os arquivos base do curso, e ao terminal da VM em tempo real, com essas ferramentas conseguiremos já instalar o Docker utilizando alguns scripts

shell vistos ao decorrer da apresentação teórica, os alunos utilizarão o terminal para rodar esses scripts para instalar o Docker com o autocomplete padrão (Docker Inc. [2026a], DigitalOcean [2026])

Usando o terminal da VM e estando no diretório padrão */minicurso*, acesse a pasta de scripts com *cd/scripts*, lá teremos acesso ao script *docker.sh* que será responsável pela instalação inicial do docker. Organizamos o conteúdo desse script da seguinte forma:

Código-fonte 46 Atualização de pacotes.

```
#!/bin/bash
$ sudo apt update -y
```

Fonte: Autoria própria.

Código-fonte 47 Instalação de pré-requisitos.

```
$ sudo apt install apt-transport-https ca-certificates curl
$ software-properties-common -y
```

Fonte: Autoria própria.

- Em seguida, adicionamos a chave GPG do repositório oficial do Docker no sistema e o repositório Docker às fontes do APT:

Código-fonte 48 Configuração do repositório Docker.

```
# Cria o diretório para armazenar chaves GPG do APT (se não existir)
sudo install -m 0755 -d /etc/apt/keyrings

# Baixa a chave GPG oficial do Docker e converte para formato binário
curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo gpg
↪  --dearmor -o /etc/apt/keyrings/docker.gpg

# Ajusta permissões para que todos possam ler a chave
sudo chmod a+r /etc/apt/keyrings/docker.gpg

# Adiciona o repositório oficial do Docker à lista de fontes do APT
echo "deb [arch=$(dpkg --print-architecture)
↪  signed-by=/etc/apt/keyrings/docker.gpg]
↪  https://download.docker.com/linux/ubuntu $(lsb_release -cs) stable" |
↪  sudo tee /etc/apt/sources.list.d/docker.list > /dev/null
```

Fonte: Autoria própria.

- Atualizamos mais uma vez os pacotes para o docker ser reconhecido:

Código-fonte 49 Atualização após adicionar repositório.

```
$ sudo apt update -y
```

Fonte: Autoria própria.

- E por fim, instalamos o Docker e já configuramos o usuário 'aluno' para utilizá-lo corretamente:

Código-fonte 50 Instalação do Docker.

```
# Verifica as versões disponíveis e a origem do pacote docker-ce
apt-cache policy docker-ce

# Instala o Docker Community Edition
sudo apt install docker-ce -y

# Adiciona o usuário atual ao grupo "docker" (permite usar sem sudo)
sudo usermod -aG docker ${USER}

# Recarrega a sessão do usuário para aplicar as permissões de grupo
su ${USER}

# Adiciona o usuário "aluno" ao grupo "docker"
sudo usermod -aG docker aluno
```

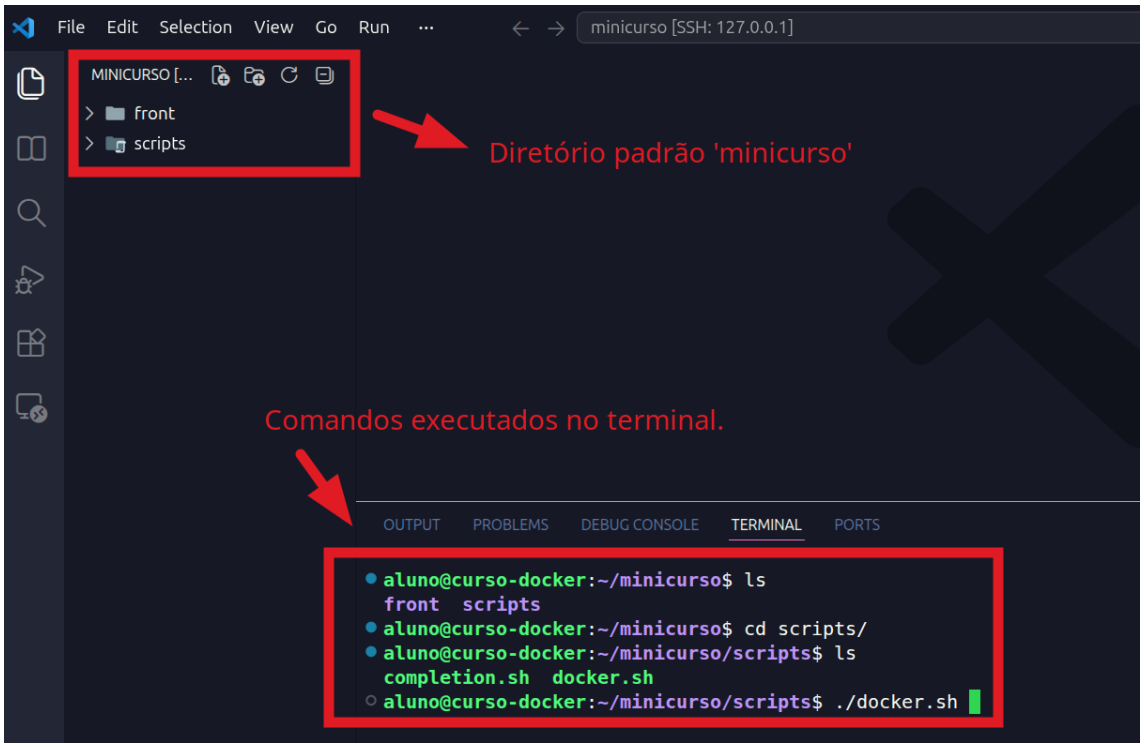
Fonte: Autoria própria.

Isso é apenas uma descrição do que o script faz para instalar o docker, agora para executar todos esses passos de uma vez, estando na pasta */scripts* rode ele pelo terminal com o comando *./docker.sh*, veja a Figura 2.6 aonde mostra os comandos usados no terminal para rodar o script. Após a execução do script confirme a instalação do docker com *docker − −version* no terminal, ao longo do processo algumas mensagens serão visíveis no terminal confirmando a configuração do docker no sistema, veja as Figuras 2.7 e 2.8.

Figura 2.7: Mensagens de sucesso na instalação do Docker e dependências.

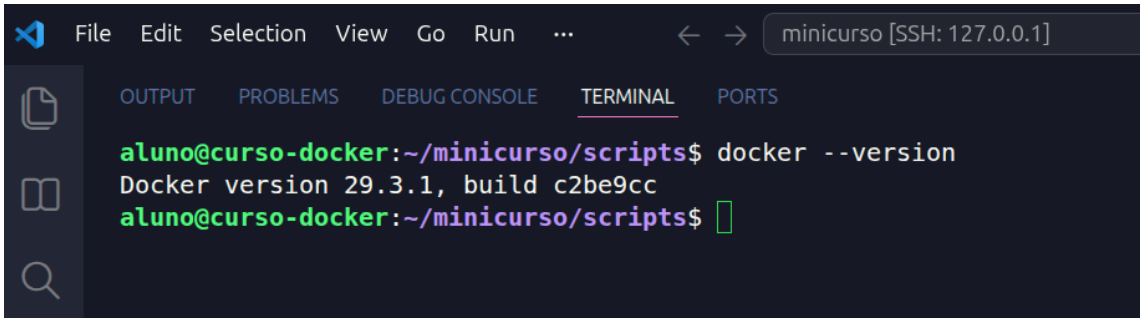
Fonte: Autoria própria.

Figura 2.6: Comandos para a execução do script de instalação do Docker.



Fonte: Autoria própria.

Figura 2.8: Comando *docker –version* executado, confirmando sucesso na instalação.



Fonte: Autoria própria.

Outra passo essencial para a instalação do docker são as configurações dos seus autocompletes de código, essa parte é totalmente *opcional*, mas recomendamos essa configuração para melhorar a usabilidade dos comandos docker no terminal. Para o autocomplete também fizemos um script de instalação automática, ele está na mesma pasta em *scripts*, e conta com o seguinte conteúdo:

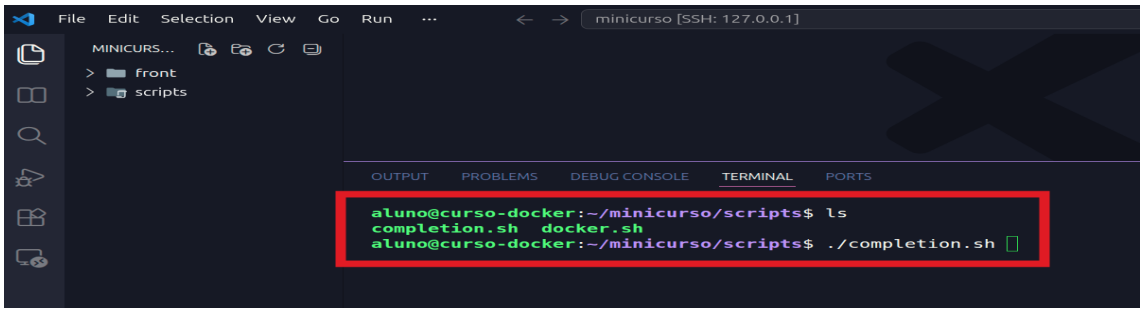
Código-fonte 51 Instalação do autocomplete.

```
#!/bin/bash

# Instala o pacote de autocompletar comandos no bash
sudo apt install bash-completion -y

# Adiciona ao .bashrc a ativação do bash-completion (se disponível)
cat <<EOT >> ~/.bashrc
if [ -f /etc/bash_completion ]; then
    . /etc/bash_completion
fi
EOT

# Cria o diretório local para armazenar scripts de autocompletar
mkdir -p ~/.local/share/bash-completion/completions

# Gera e salva o script de autocompletar do Docker
docker completion bash >
↪  ~/.local/share/bash-completion/completions/docker
```

Fonte: Autoria própria.

Aqui instalamos o pacote **bash completion**, e com um comando do Docker já configuramos esse pacote com os autocompletes padrão do Docker. Da mesma forma que foi com a instalação, rode ele pelo terminal com o comando *./completion.sh* para finalizar essa parte de configuração, veja a Figura .

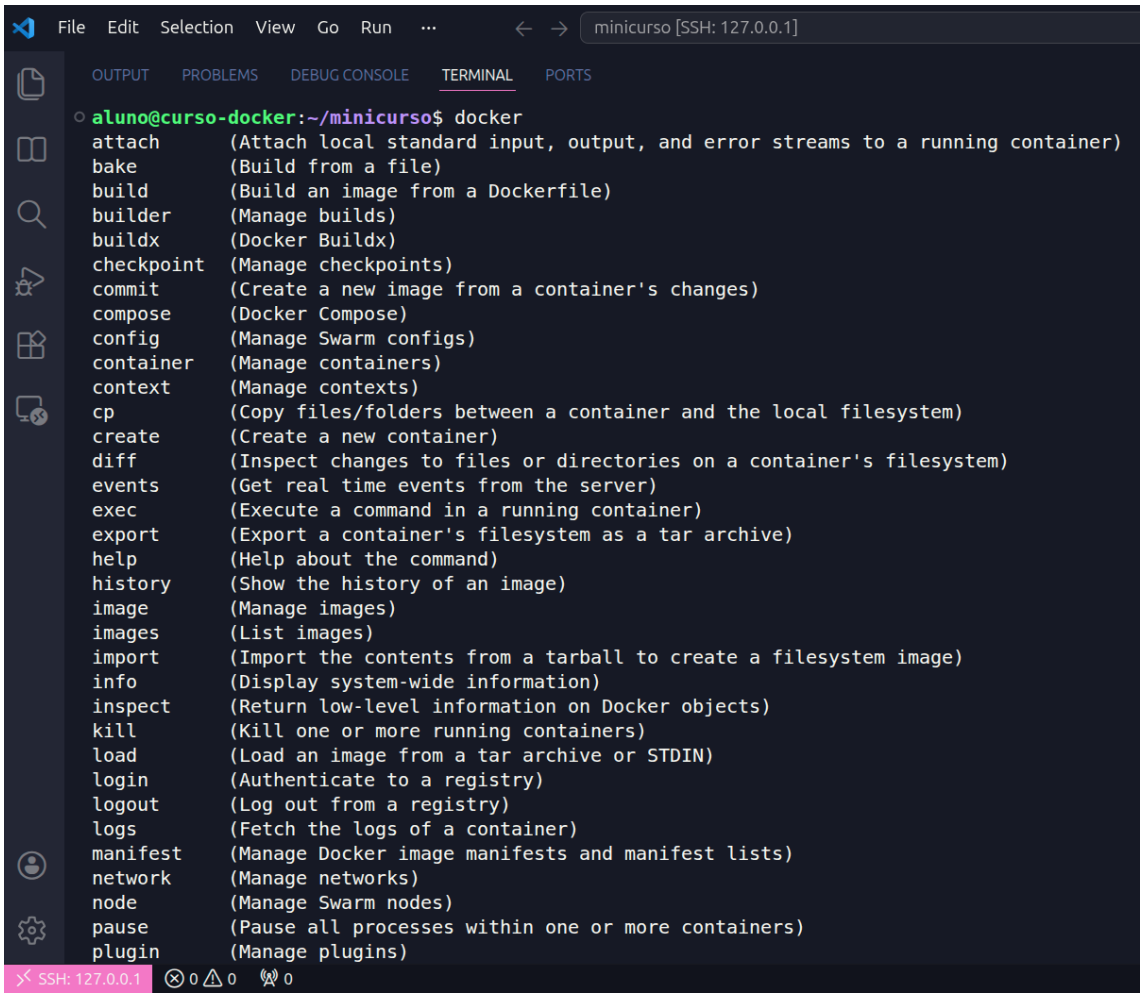
Figura 2.9: Comando no terminal para executar a instalação do autocomplete.



Fonte: Autoria própria.

Tendo sucesso com essas configurações, reinicie a VM para e conecte-se nela novamente para aplicar corretamente todas as mudanças. Após isso, você poderá ver se o autocomplete funcionou digitando **docker** no terminal e apertando a tecla **tab** tentando auto completar o comando, o resultado deverá ser uma série de comandos do **docker** com uma leve descrição sobre suas funções em cada um deles, veja a Figura 2.10.

Figura 2.10: Autocompletes do Docker no terminal após a instalação e configuração feita pelo script.



Fonte: Autoria própria.

Comandos básicos

Uma explicação e execução de alguns comandos básicos do Docker, incluindo: build, run, image, create, e pull. Nesse processo vamos baixar e construir algumas imagens, e serão criados alguns containers para introduzir o funcionamento do Docker.

Para testar os comandos básicos e logo depois mexer com o docker compose, iremos utilizar um app de exemplo da disponível pela própria documentação do Docker, trata-se de CRUD simples escrito em JavaScript com capacidade de armazenar os dados em banco local [MDN Web Docs, 2026], jogando tudo em um único arquivo *.db* em */etc/todos*, ou em um banco MySQL. Na VM que disponibilizamos já separaramos os arquivos do **app** na pasta */front*, mas dá para clonar os arquivos do git através do seguinte comando [Docker Inc., 2026b]:

Código-fonte 52 Clonando aplicação de exemplo.

```
$ git clone https://github.com/docker/getting-started-app.git
```

Fonte: Autoria própria.

A partir daqui vamos seguir com explicação tendo em mente que a pasta principal será a */front*. Então, como primeiro passo vamos mostrar como construir uma imagem Docker a partir dos arquivos desse **app**, para isso é necessário o Dockerfile para o build da imagem. Primeiro criamos um arquivo chamado Dockerfile na pasta **front** da seguinte forma:

Código-fonte 53 Dockerfile.

```
FROM node:24-alpine
WORKDIR /app
COPY . .
RUN npm install --omit=dev
CMD ["node", "src/index.js"]
EXPOSE 3000
```

Fonte: Autoria própria.

Esse Dockerfile tem como função usar node:24-alpine (distro Linux com node.js pré-instalado), usa */app* como a pasta de trabalho, copia o código fonte para a imagem, instala as dependências e faz o app escutar a porta 3000, de acordo com as configurações do próprio app. Feito isso, conseguimos fazer o build da imagem com o seguinte comando docker no terminal:

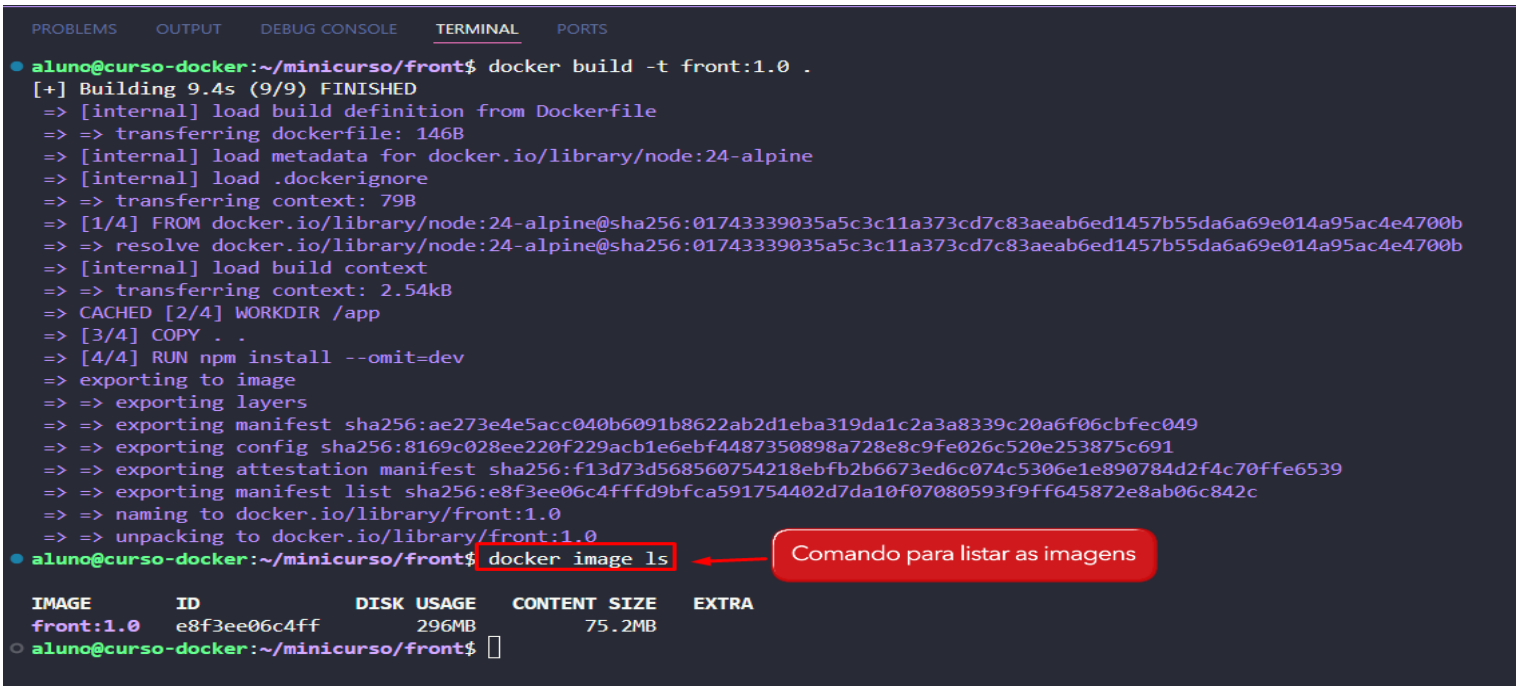
Código-fonte 54 Build.

```
# Chamamos docker build no terminal para o build, -t para especificar o
↪  nome e a tag da imagem, e . para indicar o caminho na VM aonde esta o
↪  Dockerfile que queremos buildar
docker build -t front:1.0 .
```

Fonte: Autoria própria.

Após rodar o comando, a seguinte mensagem de sucesso da Figura 2.11 é esperada no terminal, você também consegue confirmar o build com o comando ***docker image ls***, que lista as imagens do sistema:

Figura 2.11: Build da imagem no terminal + Listagem das imagens do sistema.



Fonte: Autoria própria.

Com o build feito já conseguimos rodar o primeiro container deste minicurso, para isso vamos usar a imagem que acabamos de construir e o comando ***docker run*** para subir o container. No comando também precisamos especificar a porta em que vamos acessar de maneira externa o app, para isso usamos a flag -p e os argumentos **(PORTA ACESSO EXTERNO):(PORTA ACESSO INTERNO DO CONTAINER)**, lembrando que esse app roda na porta 3000, o comando então fica dessa forma:

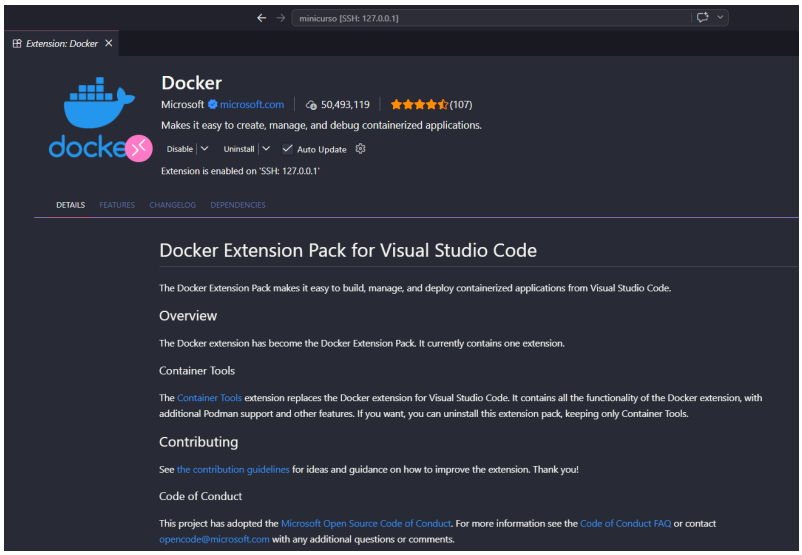
Código-fonte 55 Execução com docker run.

```
1 # Aqui também passamos a flag -d para o container rodar em segundo plano
  ↪  no terminal.
2 docker run -d -p 3000:3000 front:1.0
```

Fonte: Autoria própria.

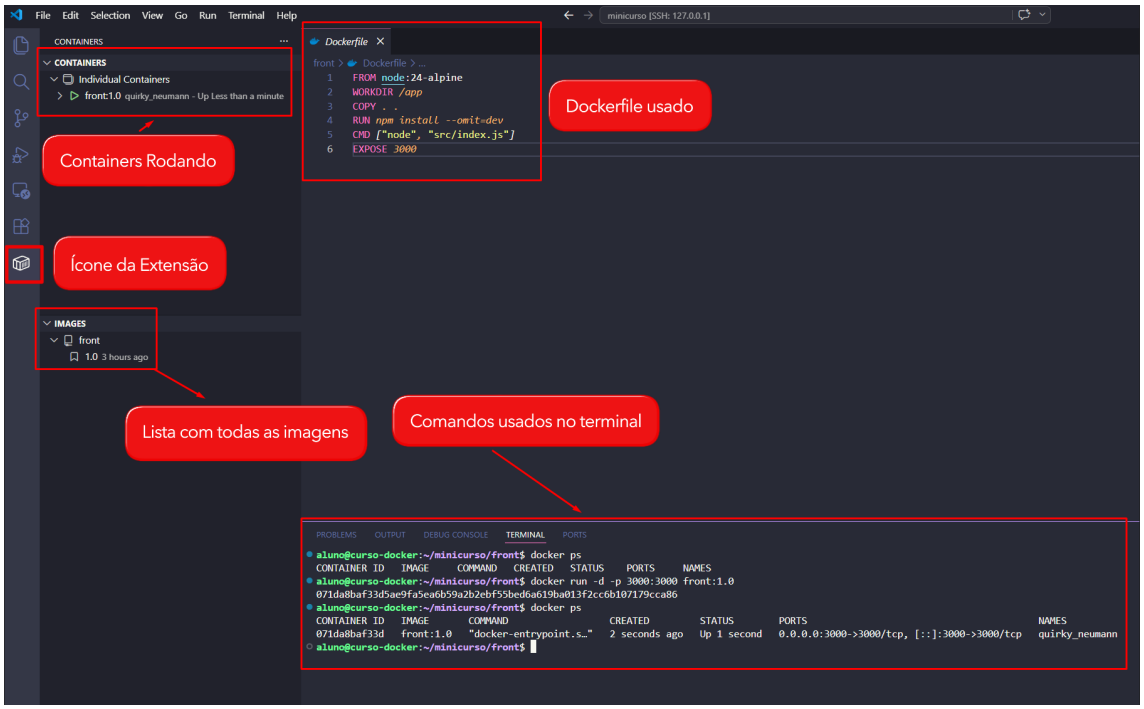
A partir daqui, para visualizar e gerenciar melhor os recursos docker vamos utilizar mais uma extensão do VsCode chamada Docker da Microsoft, veja a Figura 2.12. Ela nos permite gerenciar o recursos do Docker como imagens, container, volumes, network, de maneira mais fácil, e melhora a visualização dos recursos e containers em execução. Então após rodar o comando, acesse no painel esquerdo a extensão do docker para visualizar o container rodando e a imagem já presente no sistema, veja a Figura 2.13.

Figura 2.12: Extensão Docker da Microsoft.



Fonte: Autoria própria.

Figura 2.13: Comandos para visualizar os containers + Extensão Docker no VsCode.

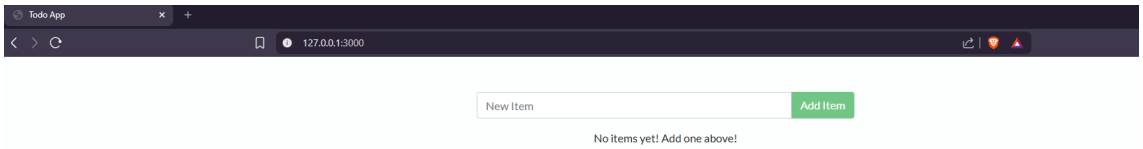


Fonte: Autoria própria.

Agora que subimos a aplicação já conseguimos acessá-la através da porta 3000 no nosso localhost, acessando no navegador temos a seguinte interface da Figura 2.14. Aqui você está livre para inserir, atualizar e excluir a lista de dados para testar a aplicação. Teste também os comandos ***docker stop (nome-do-container)*** e ***docker start (nome-do-container)***, para parar e iniciar novamente a aplicação, use o **docker ps** para verificar o nome do container antes de rodar os comandos, você deve notar que os dados adicionados ao CRUD não serão excluidos nesse processo e que nome do container permanece o mesmo, contudo é um nome gerado

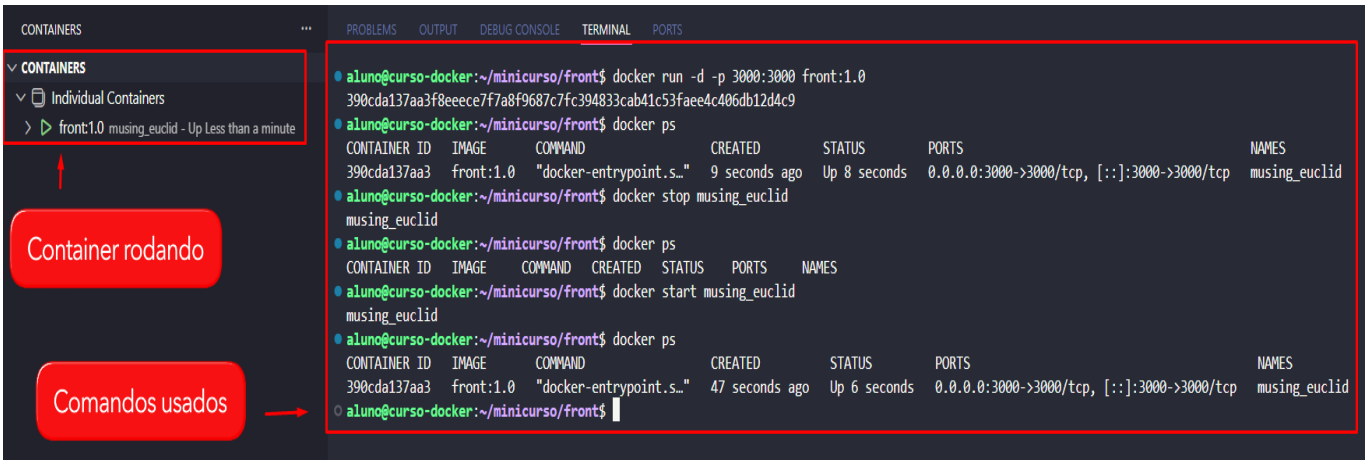
aleatoriamente já que não especificamos nenhum na criação, veja a Figura 2.15.

Figura 2.14: Interface do Todo App.



Fonte: Autoria própria.

Figura 2.15: Comandos Start e Stop.



Fonte: Autoria própria.

Contudo mesmo parecendo que esses dados são persistentes na verdade eles não são. Expiremente parar o container em execução e removê-lo com ***docker rm (nome-do-container)***, ao subir o container novamente com ***docker run*** e acessar a interface verá que os dados sumiram. Isso acontece porque não especificamos nenhuma forma de armazenar esses dados ao subir o container, então por padrão ele armazena dentro do próprio container, e quando removemos aquele container, os dados vão junto consequentemente.

Para evitar isso vamos criar o recurso chamado **volume** dentro do docker, capaz de armazenar dados dos containers, e aprimorar nosso comando ***docker run*** com alguns argumentos que vão permitir a persistência desses dados e a criação de um container com o nome personalizado. Para isso, basta rodar a sequência de

comandos abaixo, aonde criamos o volume e já subimos o novo container com um nome personalizado:

Código-fonte 56 Criação de volume + Execução do container com o volume.

```
1 docker volume create db
2 # Pare e remova o app container com docker rm -f
  ↪ <nome-do-container-atual>, e inicie novamente com:
3 docker run -d -p 3000:3000 --mount type=volume,src=db,target=/etc/todos
  ↪ --name frontend front:1.0
```

Fonte: Autoria própria.

Agora mesmo que você adicione vários dados ao CRUD e depois remova esse container, ao subir ele novamente usando esse mesmo mapeamento da flag $--mount$, os dados não irão desaparecer e estarão persistentes dentro da VM, desde que você não exclua esse ***volume db*** que criamos com o Docker.

Prática usando Docker Compose

Nessa parte final do minicurso, serão criados arquivos Compose para cada serviço que testamos anteriormente, no processo iremos testar e modificar alguns parâmetros, incluindo a configuração de Environments, persistência de dados usando um banco MySQL, mudar a exposição de portas de cada aplicação, e atribuir stacks a uma network. Todas essas etapas tem como objetivo testar comandos do Compose e mostrar a praticidade do uso dessa ferramenta.

Como explicado anteriormente na seção 2.5, o Docker Compose serve para facilitar de várias formas o uso do Docker em qualquer ambiente que esteja com ele instalado. Como os arquivos compose funcionam através de definições escritas em arquivos manifesto (***.yaml***), a configuração de serviços ficam visualmente melhor de entender, mais fácil de replicar em outros ambientes e de gerenciar os serviços rodando.

Para iniciar com essa parte prática, vamos ver primeiro como fica a definição do serviço frontend que acabamos de subir com um volume prórpio. Na pasta **front** crie um arquivo chamado ***compose.yaml***, o nome deve ser exatamente esse para que os comandos do docker compose funcionem corretamente. Nele você vai começar definindo os serviços no geral e abaixo dessa definição, cada serviço de modo individual, em um arquivo compose podemos fazer várias definições que personalizam cada serviço, contudo devemos sempre tomar cuidado com a indentação dessas configuração, por se tratar de arquivo ***yaml*** a indentação é bem importante para cada configuração.

Após criar o arquivo, traduziremos da seguinte forma o comando visto na Figura 56. Definimos o primeiro serviço como **front**, em suas configurações usamos

container-name, **ports**, **image** e **volumes** para definir tudo que usamos no comando e preenchemos de acordo, e no final do arquivo como estamos referenciando um volume criado por um comando Docker, fora da indentação de **services** definimos os **volumes** que serão usados, veja abaixo a Figura 57 como ficou o conteúdo completo do arquivo compose:

Código-fonte 57 Primeira versão do compose de frontend.

```
services:
  front:
    container_name: frontend
    ports:
      - 3000:3000
    image: front:1.0
    volumes:
      - db:/etc/todos

volumes:
   db:
     name: db
     driver: local
```

Fonte: Autoria própria.

Para subir o serviço usando docker compose e o arquivo que acabamos de configurar, pare o container frontend se ele estiver rodando e estando na pasta **front** digite o comando abaixo no terminal:

Código-fonte 58 Comando para subir serviços com o Docker Compose.

```
# Usamos -d para rodar em segundo plano
docker compose up -d
```

Fonte: Autoria própria.

Ao subir novamente você deverá notar que os dados que você inseriu ao CRUD ainda permanecem lá, caso você não tenha excluido o **volume db** anteriormente, isso acontece porque não mudamos algum parâmetro da aplicação em si, apenas mudamos o modo como a **configuramos** e **subimos** ela no servidor. Há outras formas também de persistir dados dos containers sem precisarmos de definir um volume antes, no mesmo arquivo compose vá até a configuração de volumes do serviço front e aponte para uma pasta qualquer do servidor e lá no final do arquivo remova a configuração volumes externos. Para esse formato definimos no arquivo exemplo o caminho ***./data***, aonde **.** refere-se a pasta aonde está o compose e ***data*** a pasta de destino, não é necessário criar a pasta ***data*** previamente assim que subirmos o serviço ela será criada se não existir. Ficando da seguinte forma a nova versão do arquivo compose, veja abaixo a Figura 59 :

Código-fonte 59 Segunda versão de frontend com mudança no volume.

```
services:
  front:
    container_name: frontend
    ports:
      - 3000:3000
    image: front:1.0
    volumes:
      - ./data:/etc/todos
```

Fonte: Autoria própria.

Rode novamente o comando da Figura 58 na pasta **front**, ao subir você deverá notar que a pasta **data** foi criada no processo e dentro dela estará um novo arquivo ***todo.db***, que a partir de agora armazenará o conteúdo que você inserir no CRUD da mesma forma que era com o volume, porém de modo local na própria pasta.

Também conseguimos relacionar serviços de **stacks** (arquivos compose) diferentes utilizando o docker compose. Para demonstrar isso vamos criar um novo arquivo compose e subir um container do banco MySQL a fim de usar ele para armazenar nossos dados do CRUD em tabelas, função que esse frontend já permite através de algumas configurações. Antes de tudo, precisamos arrumar uma forma com que nossos containers possam comunicar entre si sem depender do ip do servidor, para essa finalidade o docker possui um recurso chamado **network**, esse recurso nada mais é que uma rede interna do docker aonde conseguimos colocar containers como se fossem máquinas idenpendentes dentro dessa rede para que assim consigam interagir de modo mais fácil, normalmente para cada **stack** que criamos automaticamente o docker já cria uma network aonde todos os containers dentro da **stack** já participam dessa rede, mas temos a opção também de criar uma network e selecionar a dedo quais containers irão participar dela, dessa forma criando diferentes ambientes isolados para aplicações. Para criar a network basta digitar o seguinte comando no terminal:

Código-fonte 60 Comando para criar uma network Docker.

```
# Vamos chamar nossa network de frontend para facilitar
docker network create frontend
```

Fonte: Autoria própria.

Com a network criada vamos voltar na pasta **minicurso** e criar a pasta **mysql**, nela que vamos configurar o container do MySQL. Para o arquivo compose seguiremos o mesmo modelo que fizemos anteriormente, primeiro definindo os serviços, o nome do container, a imagem que será utilizada e o volume que persistiremos

na própria pasta, a novidade nessa configuração vem com a adição de **environments** que são as variárieis que podemos passar para a aplicação com o objetivo de configurá-la, para as variáveis passaremos a senha do usuário **root** no banco e um nome para o **database** inicial, por último para colocar esse container na network que criamos, vamos adicionar no final do arquivo o parâmetro **networks**, e como **default** da **stack** colocamos o nome da network e a flag **external** habilitada. Ficando dessa forma como mostra a Figura 61, lembrando que o caminho de **volumes** e o nome das variáries são todos de acordo com a documentação do próprio MySQL:

Código-fonte 61 Configuração para subir o MySQL + Network.

```
services:
  mysql:
    container_name: mysql-db
    image: mysql:8.0
    volumes:
      - ./data:/var/lib/mysql
    environment:
      MYSQL_ROOT_PASSWORD: curso
      MYSQL_DATABASE: frontdb

networks:
  default:
    name: frontend
    external: true
```

Fonte: Autoria própria.

Estando na pasta **MySQL** no terminal, suba esse container com o comando da Figura 58. Com isso já temos o banco rodando, agora para conectá-lo ao nosso **frontend** vamos precisar usar algumas variáveis específicas desta aplicação e adicioná-las no seu arquivo compose, as variáveis necessárias se encontram definidas em **front/src/persistence** no arquivo **mysql.js**. Além de definir as variáveis vamos adicionar a **network** ao final do arquivo compose, para que o **frontend** e o **banco** possam se comunicar, ao final das alterações o arquivo compose deve ficar da seguinte forma como mostra a Figura 62.

Código-fonte 62 Terceira e última versão do app com a Network e conexão com MySQL.

```
services:
  front:
    container_name: frontend
    ports:
      - 3000:3000
    image: front:1.0
    environment:
      MYSQL_HOST: mysql
      MYSQL_USER: root
      MYSQL_PASSWORD: curso
      MYSQL_DB: frontdb

networks:
  default:
    name: frontend
    external: true
```

Fonte: Autoria própria.

Ao subir novamente, o frontend já estará configurado para usar o MySQL e já estará gravando os dados do CRUD em tabelas dentro do banco (Depois falar como consultar essas informações nas tabelas).

No ambiente corporativo, a medida que o número de aplicações e recursos Docker vão crescendo, fica cada vez mais complexo gerenciar todos esses recursos. Por conta disso vamos ensinar a como subir um ferramenta que nos auxilia bastante em ambientes com o Docker instalado. Estamos falando do **Portainer**, ele é uma plataforma de gerenciamento leve que simplifica a administração de contêineres Docker, Docker Swarm e Kubernetes. Ele elimina a necessidade de comandos complexos no terminal, permitindo criar, monitorar e gerenciar contêineres, imagens, redes e volumes via navegador web.

Para finalizar o conteúdo do minicurso vamos configurar o compose dele, na pasta **minicurso** crie a pasta **portainer**, nela vamos configurar o compose da seguinte forma, primeiro definindo a imagem usada pelo portainer, vamos definir a porta **9000** para acesso externo via **localhost** e alguns volumes padrões indicados pela própria documentação do Portainer, ficando dessa forma o compose conforme mostra a Figura 63 [Portainer.io, 2026].

Código-fonte 63 Último Compose do minicurso, subindo o Portainer.

```yaml
services:
  portainer:
    image: portainer/portainer-ce
    container_name: portainer
    ports:
      - 9000:9000
    volumes:
      - ./data:/data
      - /var/run/docker.sock:/var/run/docker.sock
```

Fonte: Autoria própria.

2.7 Declaração de uso de IA generativa

☑ NÃO utilizei ferramentas de IA generativa para produzir, reescrever ou expandir trechos do texto do meu mini-curso. Eventuais correções automáticas de ortografia/gramática do editor de texto não se enquadram nesta declaração.

☐ UTILIZEI ferramentas de IA generativa de forma ética e transparente, exclusivamente como apoio redacional (ex.: revisão gramatical, melhoria de clareza, sugestão de organização), sem terceirização do conteúdo intelectual. Declaro que: (a) informei o uso; (b) revisei integralmente e sou responsável(a) pelo texto final; (c) conferi a veracidade de dados, citações e referências, garantindo que não há referências inventadas, trechos sem fonte ou conteúdo não verificável.

Se marcou acima a opção UTILIZEI, descreva de forma objetiva:

- **Ferramenta(s)/versão(ões):** NÃO SE APLICA;
- **Seções/trechos impactados:** NÃO SE APLICA;
- **Finalidade(s):** NÃO SE APLICA.

2.8 Bibliografia

DigitalOcean (2026). How to install and use docker on ubuntu 22.04. Disponível em: https://www.digitalocean.com/community/tutorials/how-to-install-and-use-docker-on-ubuntu-22-04. Acesso em: 3 maio 2026.

Docker Inc. (2026a). Docker documentation. Disponível em: https://docs.docker.com/. Acesso em: 3 maio 2026.

Docker Inc. (2026b). Getting started app. Disponível em: https://github.com/docker/getting-started-app. Acesso em: 3 maio 2026.

GOMES, R. (2016). *Docker para Desenvolvedores*. Casa do Código. Disponível em: https://www.casadocodigo.com.br/products/livro-docker. Acesso em: 24 maio 2026.

MDN Web Docs (2026). Crud. Disponível em: https://developer.mozilla.org/pt-BR/docs/Glossary/CRUD. Acesso em: 3 maio 2026.

Portainer.io (2026). Install portainer ce with docker on linux. Disponível em: https://docs.portainer.io/start/install-ce/server/docker/linux. Acesso em: 3 maio 2026.

Minicurso 3

Introdução ao GameDev com Python

Arthur Barbosa Lunas, Daniel Alves de Sousa

Resumo

Este minicurso apresenta uma introdução ao desenvolvimento de jogos digitais utilizando a linguagem Python e a biblioteca Pygame. A atividade combina conteúdos teóricos e práticos, abordando conceitos fundamentais de Game Development, as principais áreas envolvidas na criação de jogos e noções sobre a indústria de jogos digitais. Além disso, são apresentados os procedimentos para configuração do ambiente de desenvolvimento com Linux, Venv e Pygame. Na etapa prática, os participantes desenvolvem um jogo do tipo Snake, aplicando conceitos como criação de janelas gráficas, manipulação de eventos, movimentação de personagens, detecção de colisões e renderização de elementos na tela. Ao final do minicurso, espera-se que os participantes compreendam os fundamentos do desenvolvimento de jogos 2D e sejam capazes de criar projetos simples utilizando Python e Pygame.

3.1 Introdução

Neste minicurso de Introdução ao Game Development com Python e Pygame, o objetivo é apresentar ao público os fundamentos do desenvolvimento de jogos digitais utilizando a linguagem Python 3 e a biblioteca Pygame. Busca-se proporcionar uma experiência prática e didática, na qual os participantes aprenderão os conceitos essenciais de criação de jogos 2D por meio do desenvolvimento de um projeto funcional baseado nas práticas fundamentais do Game Development.

Ao final do minicurso, espera-se que os participantes sejam capazes de compreender os principais conceitos envolvidos no desenvolvimento de jogos e implementar um jogo simples utilizando Python e Pygame.

Público Alvo

O presente minicurso será voltado para o público iniciante em programação que tenha interesse na área de desenvolvimento de jogos eletrônicos. Também pode ser útil para estudantes da área da computação que desejam ter um primeiro contato com os conceitos fundamentais do Game Development.

Pré-Requisitos (Instalação)

Para acompanhar o minicurso, recomenda-se que os participantes possuam instalados em seus computadores o Python (3.12.3), a biblioteca Pygame (2.6.1), a biblioteca Venv para criação de ambientes virtuais, um sistema operacional Linux, acesso ao terminal Linux e um editor de texto ou IDE.

Pré-Requisitos (Conhecimento)

Recomenda-se que os participantes possuam conhecimentos básicos em programação com Python, tais como variáveis, estruturas condicionais, laços de repetição e funções.

Tabela 3.1: Roteiro do Minicurso.

Módulos	**Conteúdo**	**Tempo**
Teórico	Conceito de Game Development	5 minutos
	Áreas envolvidas no desenvolvimento de jogos	6 minutos
	Principais empresas de GameDev	2 minutos
	O que é Pygame	4 minutos
	Boas práticas no desenvolvimento com Pygame	4 minutos
Prático	Criação do ambiente virtual VENV	4 minutos
	Inicialização da tela do jogo	8 minutos
	Inicialização das variáveis do jogo	8 minutos
	Manipulação de eventos (controle via teclado)	10 minutos
	Movimentação da cobra	10 minutos
	Detecção de colisões	10 minutos
	Crescimento da cobra	10 minutos
	Renderização e finalização do jogo	9 minutos

Fonte: Autoria própria.

Após a apresentação do roteiro do minicurso na Tabela 3.1, cada atividade será desenvolvida de forma progressiva, buscando apresentar os conceitos de maneira clara e acessível para os participantes.

3.2 Conceito de Game Development

Quando falamos em Game Development, ou desenvolvimento de jogos, estamos entrando em um universo que vai muito além de apenas programar.

Criar um jogo é, na verdade, construir uma experiência. Para isso, precisamos combinar diferentes áreas do conhecimento: programação, arte, design, som e até psicologia. É justamente essa mistura que torna o desenvolvimento de jogos tão interessante e também desafiador.

Ao longo deste minicurso, você vai perceber que desenvolver um jogo não começa no código. Tudo inicia com uma ideia: qual experiência queremos proporcionar ao jogador? A partir disso, definimos regras, mecânicas e objetivos. Em seguida, criamos protótipos simples para testar essas ideias antes de evoluir para algo mais completo.

Diferente de outros tipos de software, onde o foco está na eficiência ou na resolução de problemas práticos, os jogos têm um objetivo central: engajar o jogador. Isso acontece por meio da interatividade. O jogo apresenta um desafio, o jogador toma decisões, e o sistema responde imediatamente. Esse ciclo contínuo é o que mantém a atenção e cria a sensação de imersão.

Outro ponto importante é entender que cada detalhe importa. O código define como o jogo funciona, mas os elementos visuais, os sons e até pequenas animações influenciam diretamente na experiência do jogador. Por isso, desenvolver jogos é equilibrar lógica e criatividade o tempo todo.

Além disso, é interessante observar que o Game Development não surgiu de forma repentina, mas evoluiu ao longo do tempo com o avanço da tecnologia. Um dos primeiros jogos digitais foi o *Tennis for Two*, criado por William Higinbotham em 1958 (Figura 3.1a). O jogo foi desenvolvido utilizando um osciloscópio como dispositivo de exibição, permitindo simular uma partida de tênis de forma bastante simples para os padrões atuais [Brookhaven National Laboratory, 1958].

Poucos anos depois, surgiu *Spacewar!*, desenvolvido por Steve Russell e sua equipe no MIT (Figura 3.1b). Esse jogo foi executado em um computador PDP-1, uma das primeiras máquinas interativas da época, e já apresentava elementos mais avançados, como controle em tempo real, física simplificada e interação entre jogadores. Cada jogador podia movimentar a nave, acelerar e atirar enquanto tentava evitar a gravidade de uma estrela localizada no centro da tela, tornando a experiência mais dinâmica e desafiadora. Mesmo sendo um jogo simples para os padrões atuais, *Spacewar!* influenciou diversos jogos modernos e contribuiu significativamente para a evolução da indústria dos videogames [Computer History Museum, 1962].

Esses projetos foram fundamentais para estabelecer as bases do desenvolvi-

mento de jogos digitais, demonstrando como limitações tecnológicas influenciaram diretamente o design e a implementação dos primeiros jogos. Mesmo com recursos extremamente limitados, esses desenvolvedores conseguiram criar experiências interativas, conceito que permanece central no Game Development até os dias atuais.

Figura 3.1: Criadores dos primeiros jogos digitais.

(a) William Higinbotham

(b) Steve Russell

Fonte: Adaptado de Atomic Heritage Foundation [2026] e Wikipedia Contributors [2026].

Ao final, podemos pensar no *Game Development* como a capacidade de transformar ideias abstratas em mundos interativos onde o jogador não apenas observa, mas participa ativamente.

3.3 Áreas Envolvidas no Desenvolvimento de Jogos

O desenvolvimento de jogos digitais é, por natureza, um processo multidisciplinar que envolve a integração de diferentes áreas do conhecimento. Cada uma dessas áreas contribui de forma específica para a construção do produto final, tornando possível a criação de experiências interativas completas. Entre as principais, destaca-se a programação, responsável pela implementação da lógica do jogo, incluindo regras, sistemas de interação, física e comportamento dos elementos no ambiente virtual.

Paralelamente, o Game Design atua na definição da experiência do jogador, estabelecendo mecânicas, objetivos, progressão e desafios. A área de arte é responsável pela construção visual do jogo, desenvolvendo personagens, cenários, interfaces e animações que contribuem para a identidade estética e a imersão. Já o design de áudio complementa essa experiência por meio de efeitos sonoros e trilhas musicais, reforçando emoções e fornecendo *feedback* ao jogador durante a interação.

Além dessas áreas, a narrativa desempenha um papel importante na construção do universo e no engajamento do jogador, especialmente em jogos orientados por história. É importante ressaltar que essas áreas não atuam de forma isolada, mas sim de maneira integrada, exigindo alinhamento constante entre aspectos téc-

nicos e criativos. Dessa forma, o desenvolvimento de jogos pode ser compreendido como um esforço colaborativo, no qual diferentes especialidades convergem para criar experiências interativas coesas e envolventes.

3.4 Principais Empresas de Game Development

O mercado de desenvolvimento de jogos digitais é composto por diversas empresas que exercem forte influência sobre a indústria global do entretenimento. Essas organizações se destacam não apenas pela criação de títulos de grande sucesso comercial, mas também pela constante inovação tecnológica, pela capacidade de atingir públicos em escala mundial e pela definição de tendências que impactam diretamente o setor. O estudo dessas empresas é essencial para compreender os padrões de produção, os modelos de negócio, as metodologias de desenvolvimento e as oportunidades profissionais existentes na área de *game development*.

Entre as principais empresas do setor, destaca-se a *Nintendo*, considerada uma das organizações mais influentes da história dos videogames. Fundada no Japão, a empresa possui uma trajetória consolidada e é responsável por franquias extremamente reconhecidas, como *Super Mario*, *The Legend of Zelda*, *Pokémon* e *Metroid*. Além disso, a Nintendo também se tornou referência no desenvolvimento de consoles, como o *Nintendo Entertainment System (NES)*, o *Wii* e o *Nintendo Switch*, que marcaram diferentes gerações de jogadores. Sua importância para a indústria está diretamente relacionada à capacidade de unir inovação, acessibilidade e forte apelo criativo em seus produtos.

Outra empresa de grande relevância é a *Electronic Arts (EA)*, fundada em 1982 nos Estados Unidos. A EA consolidou-se como uma das maiores desenvolvedoras e publicadoras de jogos eletrônicos do mundo, sendo amplamente reconhecida por franquias de enorme sucesso, principalmente no segmento esportivo, como *FIFA* (atualmente *EA Sports FC*), *Madden NFL* e *NBA Live*. Além dos jogos de esporte, a empresa também possui forte presença em gêneros como ação, corrida e simulação, com títulos como *Battlefield*, *The Sims* e *Need for Speed*. Sua atuação demonstra a importância do planejamento estratégico e do investimento em franquias de longa duração no mercado de jogos.

A *Ubisoft* também merece destaque no cenário mundial do desenvolvimento de jogos. De origem francesa, a empresa possui estúdios distribuídos em diversos países, o que evidencia a dimensão global de suas operações. A Ubisoft é conhecida por produzir jogos de grande impacto e qualidade técnica, como *Assassin's Creed*, *Far Cry*, *Watch Dogs* e *Rainbow Six*. Seus títulos frequentemente apresentam mundos abertos, narrativas complexas e experiências multiplayer, demonstrando o elevado nível de sofisticação envolvido no processo de desenvolvimento contemporâneo.

Além dessas, a *Activision Blizzard* representa uma das maiores potências do setor. Formada pela união de duas gigantes da indústria, a empresa é responsável por franquias extremamente populares, como *Call of Duty*, *World of Warcraft*, *Diablo* e *Overwatch*. Seu destaque está relacionado especialmente ao desenvolvimento de jogos online, experiências *multiplayer* em larga escala e à consolidação do mercado de jogos competitivos e *eSports*. Essa empresa exemplifica como o setor de jogos evoluiu para além do entretenimento individual, tornando-se também um ambiente social e competitivo.

A *Valve Corporation* também é frequentemente mencionada como uma das empresas mais influentes do setor. Além da criação de jogos icônicos, como *Half-Life*, *Portal* e *Counter-Strike*, a Valve revolucionou a indústria por meio da plataforma *Steam*. Essa plataforma transformou a forma como os jogos são distribuídos, comercializados e atualizados no ambiente de computadores, tornando-se uma referência mundial em distribuição digital. O impacto da Steam foi significativo para desenvolvedores independentes e grandes estúdios, ampliando o acesso ao mercado global.

Dessa forma, observa-se que essas empresas desempenham papel fundamental na evolução do desenvolvimento de jogos digitais. Cada uma, com estratégias distintas, demonstra como a integração entre criatividade, tecnologia, design e compreensão do comportamento do público é essencial para o sucesso na indústria. O estudo dessas organizações contribui para uma visão mais ampla sobre o funcionamento do mercado e sobre as competências exigidas dos profissionais da área.

3.5 Criação do ambiente virtual - VENV (Creation of Virtual Environments)

Neste módulo, será apresentada a criação de um ambiente virtual utilizando o módulo *venv* da linguagem Python [Foundation, 2026]. Ambientes virtuais permitem a criação de espaços isolados dentro do sistema operacional, nos quais é possível instalar bibliotecas e dependências específicas para cada projeto.

O uso desse recurso é fundamental para evitar conflitos entre versões de bibliotecas, garantindo que diferentes projetos possam coexistir no mesmo computador sem interferência entre si. Dessa forma, cada aplicação mantém seu próprio conjunto de dependências, preservando a integridade da instalação global do Python.

No contexto deste minicurso, o ambiente virtual será utilizado para a instalação da biblioteca *Pygame*, assegurando que todos os participantes consigam executar os exemplos propostos sem alterar as configurações do sistema das máquinas do laboratório.

Criação e configuração do ambiente virtual

O processo de criação do ambiente virtual será realizado por meio de comandos no terminal Linux, conforme apresentado no Código 64.

Código-fonte 64 Criação da Venv.

```
#!/bin/bash

$ ls
$ mkdir ~/venvs
$ cd ~/venvs/
$ mkdir game
$ cd ..
$ python3 -m venv venvs/game/
$ ls
$ cd ~/venvs/
$ ls
$ cd ~/game/
$ ls
$ cd ..
$ source ~/venvs/game/bin/activate
$ pip list
$ pip install pygame
$ pip list
$ deactivate
$ pip list
$ source ~/venvs/game/bin/activate
```

Fonte: Autoria própria.

Inicialmente, o aluno deve abrir o terminal e verificar os arquivos e diretórios disponíveis no diretório atual utilizando o comando `ls`:

```
$ ls
```

Em seguida, cria-se um diretório chamado `venvs`, destinado ao armazenamento dos ambientes virtuais:

```
$ mkdir ~/venvs
```

Após a criação, acessa-se o diretório:

```
$ cd ~/venvs/
```

Dentro dele, é criado o diretório `game`, que representará o ambiente virtual do projeto:

```
$ mkdir game
```

Em seguida, retorna-se ao diretório anterior para executar a criação do ambiente virtual com o módulo `venv`:

```
$ cd ..
$ python3 -m venv venvs/game/
```

Após a execução do comando, o ambiente virtual será criado dentro do diretório especificado. É possível verificar sua existência listando novamente os diretórios:

```
$ ls
```

Para utilizar o ambiente criado, é necessário acessá-lo e ativá-lo. Primeiramente, navega-se até o diretório correspondente:

```
$ cd ~/venvs/
$ ls
$ cd ~/game/
$ ls
```

Em seguida, retorna-se ao diretório apropriado e ativa-se o ambiente virtual:

```
$ cd ..
$ source ~/venvs/game/bin/activate
```

Uma vez ativado, o ambiente virtual passa a isolar as dependências do projeto. É possível verificar os pacotes instalados com:

```
$ pip list
```

Na sequência, realiza-se a instalação da biblioteca *Pygame*, que será utilizada ao longo do minicurso:

```
$ pip install pygame
```

Após a instalação, os pacotes podem ser listados novamente para confirmação:

```
$ pip list
```

Caso seja necessário sair do ambiente virtual, utiliza-se o comando:

```
$ deactivate
```

Após a desativação, observa-se que os pacotes instalados deixam de aparecer no ambiente global, evidenciando o isolamento proporcionado pelo *venv*:

```
$ pip list
```

Para reativar o ambiente virtual posteriormente, basta executar novamente o comando de ativação:

```
$ source ~/venvs/game/bin/activate
```

3.6 O que é Pygame?

O **Pygame** é uma biblioteca *open-source* desenvolvida para a linguagem Python [Pygame Community, 2025], amplamente utilizada na criação de jogos digitais e aplicações multimídia. Sua principal finalidade é facilitar o desenvolvimento de jogos 2D, oferecendo recursos prontos para manipulação de gráficos, sons, animações, entrada de teclado e mouse, além do gerenciamento do *loop* principal do jogo.

A biblioteca é construída sobre a *SDL (Simple DirectMedia Layer)*, uma camada responsável por abstrair o acesso aos recursos gráficos, sonoros e aos dispositivos de entrada do computador. Com isso, o desenvolvedor pode concentrar seus esforços na lógica do jogo, na jogabilidade e na experiência do usuário, sem a necessidade de lidar diretamente com detalhes de baixo nível.

Por meio do Pygame, é possível desenvolver desde jogos simples, como quebra-cabeças e jogos de plataforma, até protótipos mais elaborados que envolvem movimentação de personagens, detecção de colisões, efeitos sonoros, trilhas musicais e animações. Além disso, a biblioteca é amplamente utilizada em contextos educacionais, devido à sua sintaxe simples e acessível, sendo especialmente indicada para o aprendizado inicial de conceitos relacionados ao *Game Development*.

Entre os principais conceitos abordados com o uso do Pygame, destacam-se o **loop do jogo**, o gerenciamento de eventos, a renderização gráfica, a manipulação de

sprites e a detecção de colisões. Esses elementos são fundamentais para a construção de jogos digitais e permitem ao estudante compreender, de forma prática, como um jogo é estruturado.

A Figura 3.2 apresenta o logotipo da biblioteca Pygame, amplamente reconhecido na comunidade de desenvolvimento, enquanto sua documentação oficial constitui uma importante fonte de consulta para funções, módulos e exemplos práticos utilizados ao longo do minicurso.

Figura 3.2: Logotipo da biblioteca Pygame.

Fonte: Pygame Community (2025).

A documentação oficial do Pygame pode ser acessada online, reunindo descrições detalhadas de funcionalidades e exemplos de uso da biblioteca [Pygame Community, 2025].

3.7 Boas Práticas no Desenvolvimento com Pygame

Ao trabalhar com Pygame, aplicar boas práticas é fundamental para criar jogos mais organizados, eficientes e fáceis de manter. Um ponto crucial é estruturar corretamente o código, separando lógica do jogo, renderização e gerenciamento de eventos. Utilizar funções e classes de forma modular permite reaproveitar componentes, facilitar a depuração e tornar o projeto escalável à medida que novas funcionalidades são adicionadas.

Outra prática recomendada é o gerenciamento adequado de recursos, como imagens, sons e fontes. É importante carregar esses recursos apenas uma vez e reutilizá-los durante o jogo, evitando sobrecarga de memória e lentidão. Além disso, organizar assets em pastas específicas, com nomes claros e consistentes, facilita a manutenção do projeto e a colaboração entre diferentes desenvolvedores.

Por fim, atenção ao loop principal do jogo é essencial. Ele deve ser eficiente,

garantindo atualizações consistentes de gráficos, sons e eventos sem travamentos ou atrasos. Também é recomendado implementar controle de FPS (frames por segundo) para manter o desempenho previsível. Testar o jogo frequentemente, revisar código e documentar funcionalidades são práticas que aumentam a qualidade final, ajudam no aprendizado e facilitam o desenvolvimento de projetos em Pygame de forma profissional.

Ambiente de Desenvolvimento no Linux

Antes de iniciar o desenvolvimento de jogos utilizando a biblioteca Pygame, é fundamental apresentar o ambiente no qual os códigos serão escritos, editados e executados. Neste mini curso, será utilizado o sistema operacional **Linux**, uma plataforma amplamente empregada em ambientes acadêmicos, laboratoriais e de desenvolvimento de software. O Linux é reconhecido por sua estabilidade, segurança, flexibilidade e compatibilidade com diversas ferramentas de programação, sendo uma excelente opção para o aprendizado do desenvolvimento de jogos com Python.

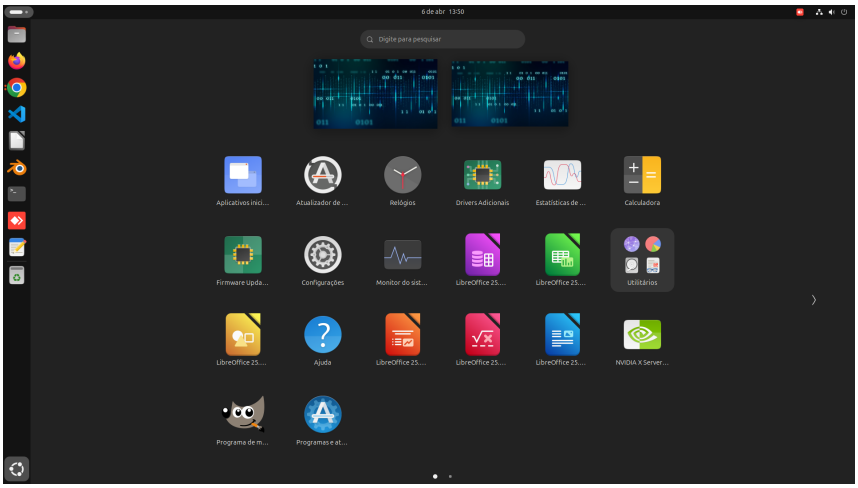
A Figura 3.3 apresenta a tela inicial do sistema operacional Linux. Este será o ambiente base em que todas as atividades práticas do mini curso serão realizadas, incluindo a criação, edição e execução dos códigos desenvolvidos com Pygame.

Figura 3.3: Tela inicial do sistema operacional Linux.

Fonte: Autoria própria.

Para iniciar a programação, o primeiro passo consiste em acessar o editor de texto que será utilizado para escrever os códigos em Python. No ambiente Linux, esse acesso pode ser realizado de duas maneiras principais. A primeira é por meio do menu de aplicativos do sistema, onde o usuário pode localizar o editor instalado, conforme ilustrado na Figura 3.4. Essa forma é bastante comum para usuários que preferem navegar pelas aplicações disponíveis no sistema.

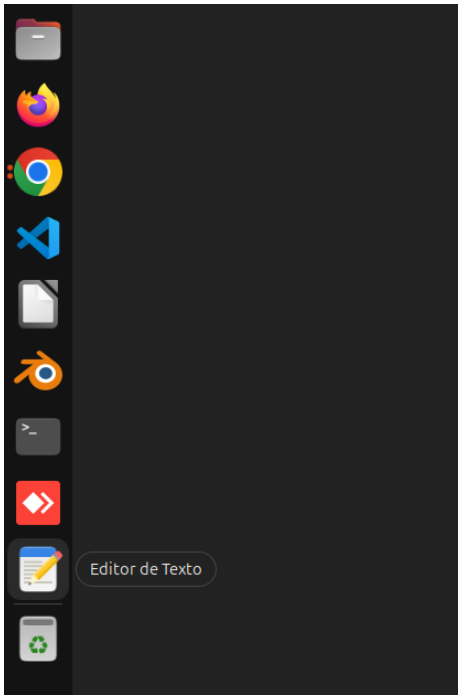
Figura 3.4: Acesso ao editor de texto pelo menu do Linux.



Fonte: Autoria própria.

A segunda forma de acesso é utilizando a barra lateral do sistema, que oferece um caminho mais rápido para abrir aplicações frequentemente utilizadas. Essa opção é apresentada na Figura 3.5, sendo uma alternativa prática para agilizar o processo de desenvolvimento.

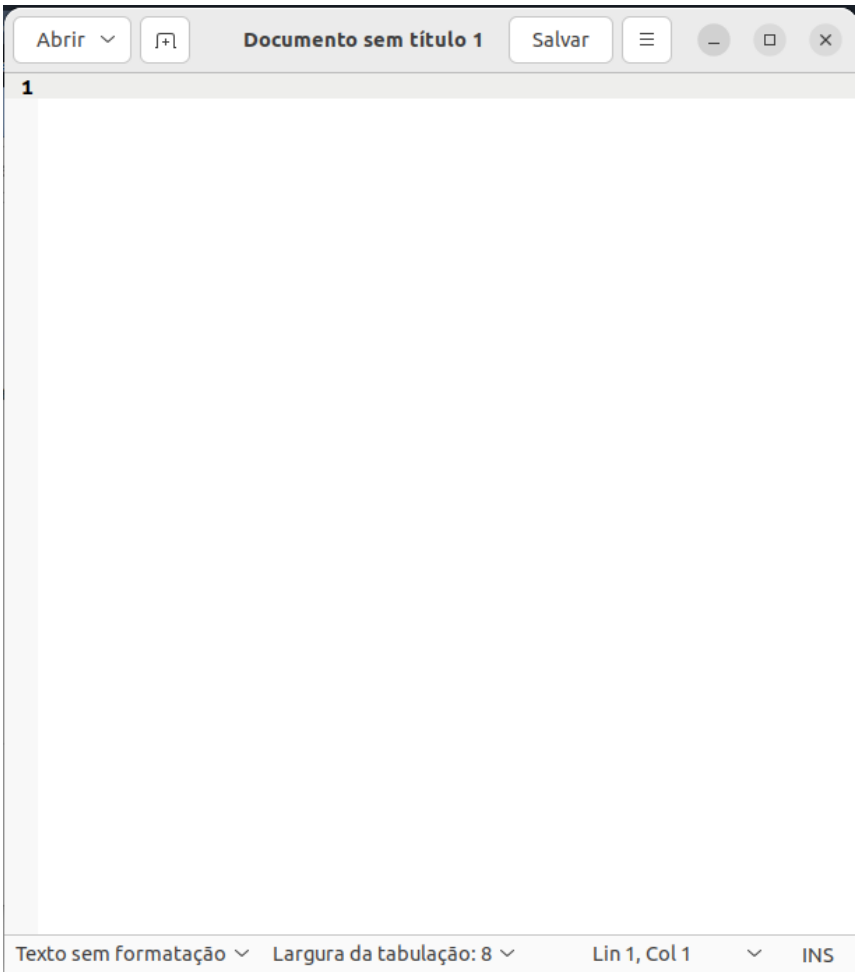
Figura 3.5: Acesso ao editor de texto pela barra lateral do Linux.



Fonte: Autoria própria.

Após a abertura do editor de texto, o ambiente inicial será apresentado vazio, conforme ilustrado na Figura 3.6. Esse espaço será utilizado para a escrita dos códigos desenvolvidos ao longo do mini curso.

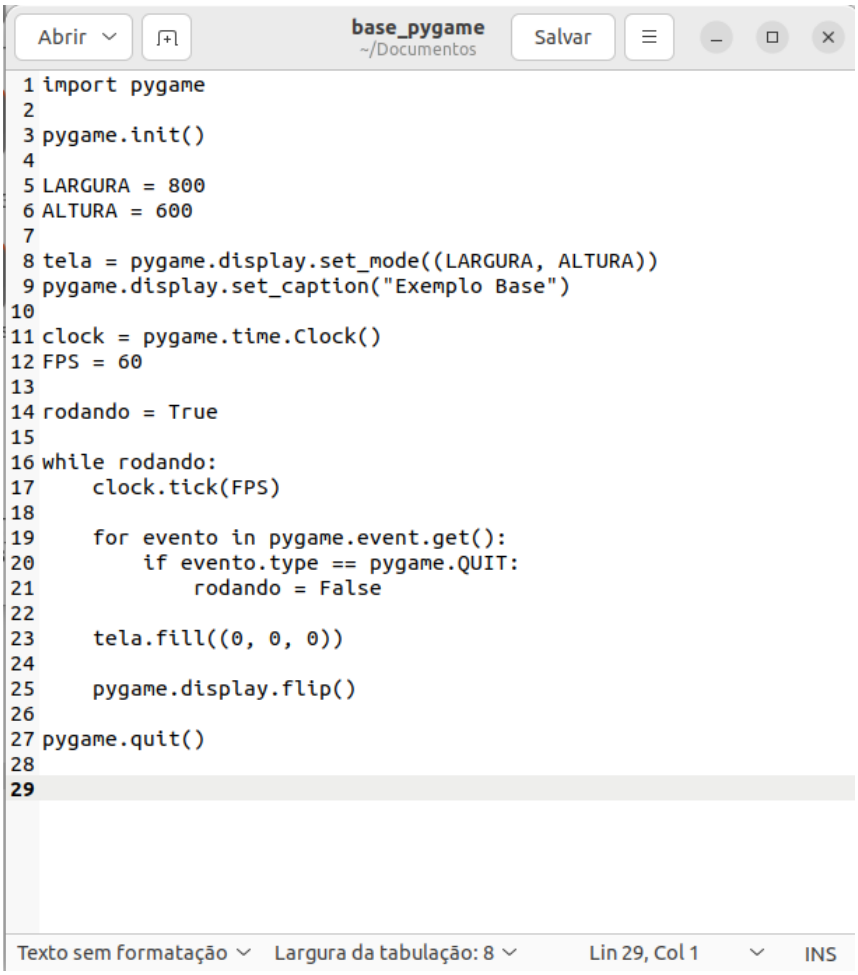
Figura 3.6: Editor de texto aberto e pronto para a escrita do código.



Fonte: Autoria própria.

Na sequência, o usuário poderá inserir o código-fonte no editor, conforme apresentado na Figura 3.7. Neste momento, o arquivo deverá ser salvo na pasta Documentos, facilitando sua localização e organização, mesmo que ainda se encontre no formato de texto padrão, sem a extensão específica da linguagem Python.

Figura 3.7: Código inserido no editor de texto em formato inicial.



Fonte: Autoria própria.

Após a escrita do código, o próximo passo consiste em salvar o arquivo, atribuindo um nome ao documento e escolhendo o local de armazenamento, como a pasta *Documentos*, por exemplo. Inicialmente, o arquivo pode ser salvo com a extensão padrão do editor de texto, como `.txt`.

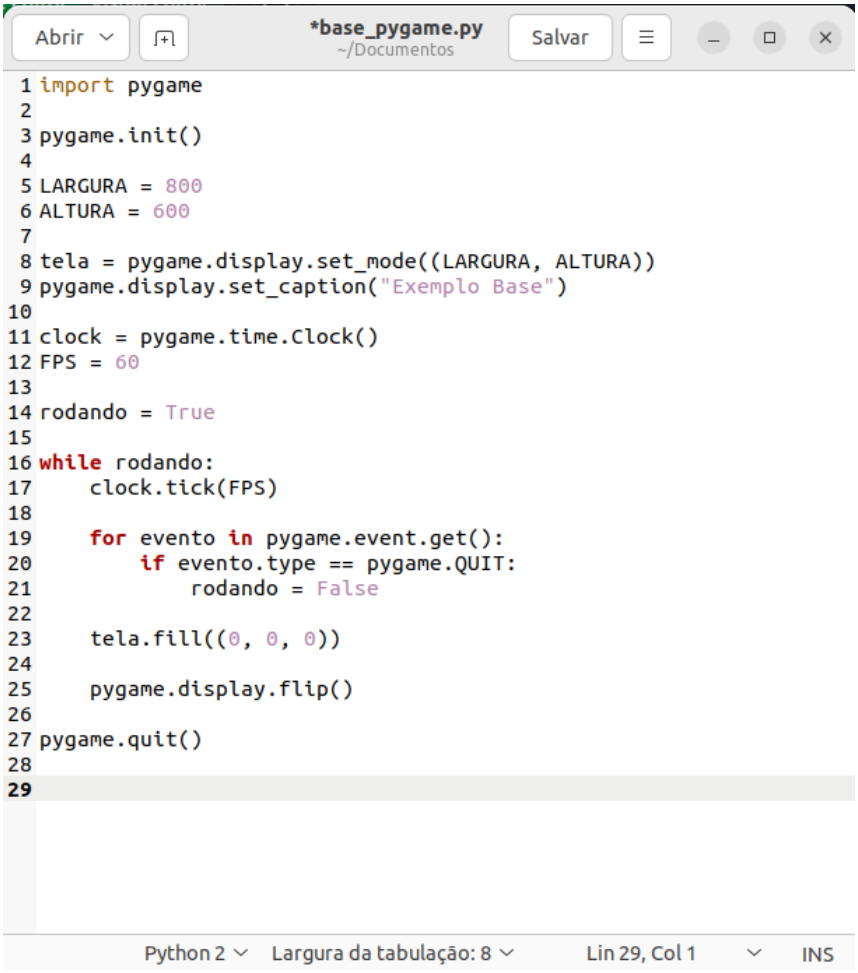
Entretanto, para que o sistema reconheça corretamente o arquivo como um script Python, é necessário alterar sua extensão para `.py`. Essa modificação pode ser realizada renomeando o arquivo diretamente no sistema operacional, substituindo a extensão original.

Após essa alteração, o arquivo passa a ser identificado como um programa em Python, permitindo sua execução no terminal e possibilitando que editores de código reconheçam automaticamente a linguagem utilizada, aplicando recursos como destaque de sintaxe e sugestões de código.

Por fim, ao abrir novamente o arquivo com a extensão correta, o editor de texto apresentará o código com destaque de sintaxe, utilizando diferentes cores para comandos, funções e estruturas da linguagem, conforme ilustrado na Figura 3.8. Esse recurso facilita a leitura e a compreensão do código, tornando o processo de

desenvolvimento mais organizado e intuitivo.

Figura 3.8: Código em Python com destaque de sintaxe no editor de texto.



Fonte: Autoria própria.

Após salvar o arquivo com a extensão `.py`, será necessário movê-lo para a pasta *game*, criada anteriormente no ambiente virtual com o *venv*. Essa prática ajuda a manter os arquivos do projeto organizados dentro da mesma pasta de desenvolvimento.

3.8 Inicialização do jogo

Antes de iniciar o desenvolvimento do jogo, é necessário definir como será estruturado o projeto prático. A seguir, apresentamos o desenvolvimento passo a passo do jogo Snake, onde cada trecho de código é acompanhado de seu resultado visual.

Inicialização da tela do jogo e controle de tempo

O primeiro passo é configurar a janela do jogo e o controle de tempo. O código abaixo cria uma janela vazia com as dimensões definidas e configura o controle de FPS (frames por segundo).

Código-fonte 65 Inicialização da tela do jogo e controle de tempo.

```
1  import pygame
2  import random
3
4  pygame.init()
5
6  LARGURA = 800
7  ALTURA = 600
8
9  tela = pygame.display.set_mode((LARGURA, ALTURA))
10 pygame.display.set_caption("Snake")
11
12 clock = pygame.time.Clock()
13 FPS = 10
```

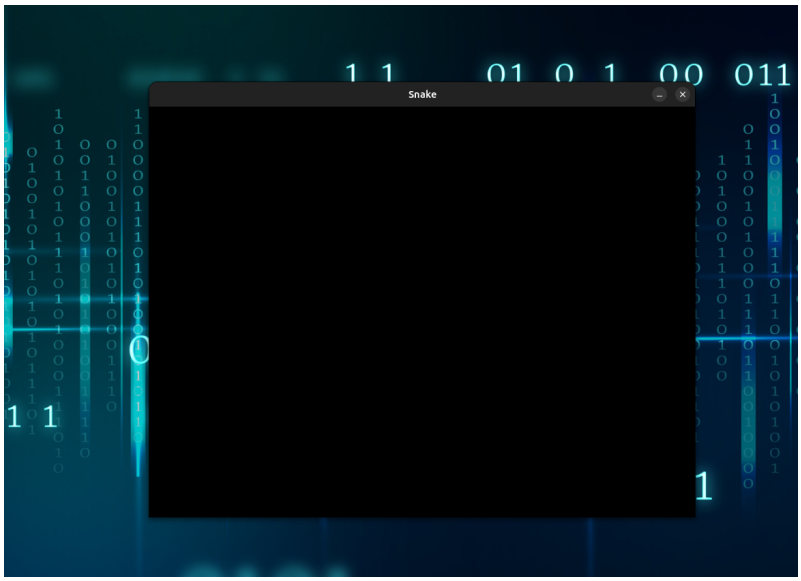
```
23 rodando = True
24 while rodando:
25     clock.tick(FPS)
26
27     for evento in pygame.event.get():
28         if evento.type == pygame.QUIT:
29             rodando = False
```

Fonte: Autoria própria.

Explicação do código:

A importação das bibliotecas necessárias para o funcionamento do jogo, sendo elas o *pygame* e o *random*, é realizada nas (Linhas 1-2). A inicialização do Pygame ocorre na (Linha 4). A definição das dimensões da janela do jogo em 800x600 pixels é feita nas (Linhas 6-7). A criação da janela gráfica e a configuração do título exibido acontecem nas (Linhas 9-10). O controle de tempo do jogo, utilizando o *clock* e a taxa de quadros por segundo (FPS), é configurado nas (Linhas 12-13). Por fim, o loop principal responsável por manter a janela aberta e capturar eventos de fechamento é implementado nas (Linhas 23-29).

Figura 3.9: Resultado: Janela vazia criada pelo Pygame.



Fonte: Autoria própria

Inicialização das variáveis

Após criar a janela, precisamos inicializar as variáveis que controlam o estado do jogo: a posição inicial da cobra, sua direção de movimento e a posição da comida.

Código-fonte 66 Inicialização das variáveis do jogo.

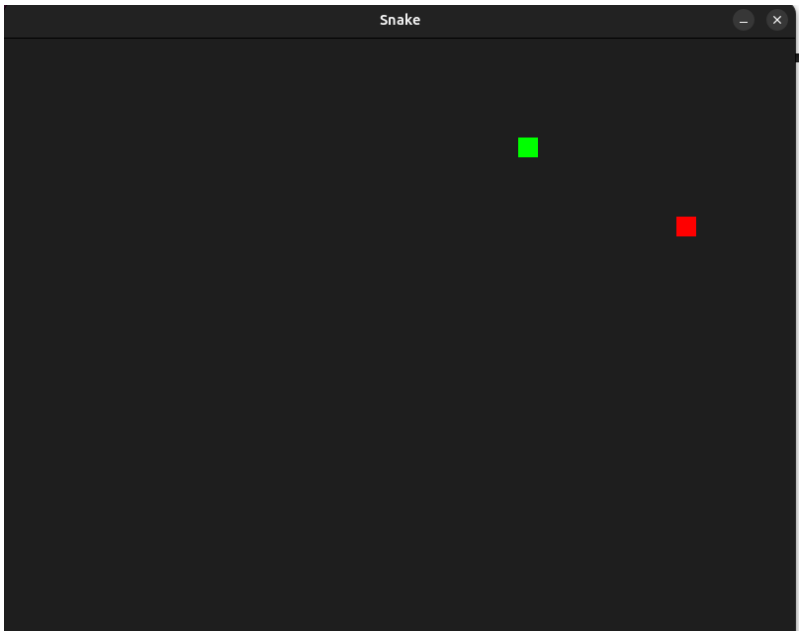
```
15 snake = [(100, 100)]
16 direcao = (20, 0)
17
18 food_pos = (
19     random.randint(0, LARGURA // 20) * 20,
20     random.randint(0, ALTURA // 20) * 20
21 )
```

Fonte: Autoria própria.

Explicação do código:

A criação da cobra como uma lista contendo sua posição inicial em (100, 100) é realizada na (Linha 15). A definição da direção inicial do movimento, configurada para deslocar a cobra 20 pixels para a direita, ocorre na (Linha 16). Já a geração da posição aleatória da comida, alinhada à grade de 20 pixels do jogo, é implementada nas (Linhas 18-21).

Figura 3.10: Resultado: Cobra (verde) e comida (vermelho) posicionadas.



Fonte: Autoria própria

Manipulação de eventos

O jogo precisa responder a eventos do usuário, como fechar a janela ou pressionar teclas para controlar a direção da cobra.

Código-fonte 67 Controle via teclado.

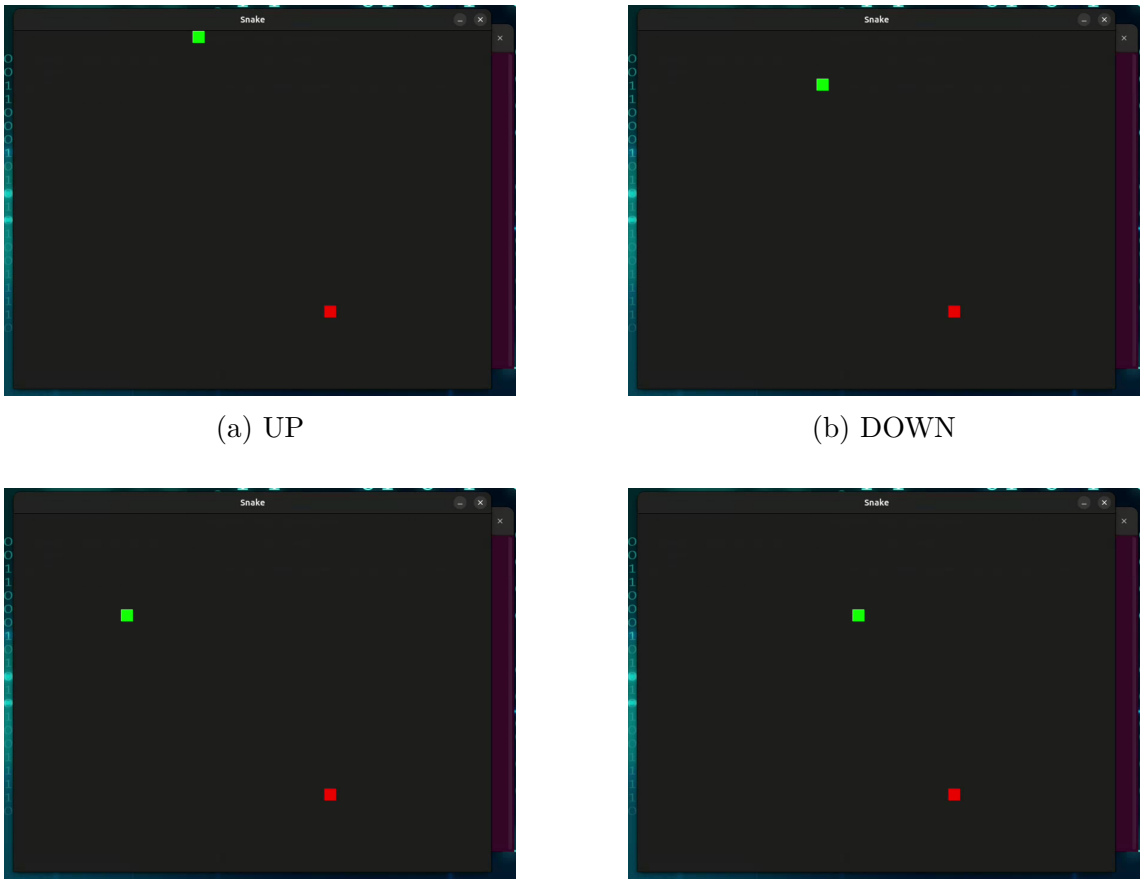
```
27      for evento in pygame.event.get():
28          if evento.type == pygame.QUIT:
29              rodando = False
30
31          if evento.type == pygame.KEYDOWN:
32              if evento.key == pygame.K_UP:
33                  direcao = (0, -20)
34              if evento.key == pygame.K_DOWN:
35                  direcao = (0, 20)
36              if evento.key == pygame.K_LEFT:
37                  direcao = (-20, 0)
38              if evento.key == pygame.K_RIGHT:
39                  direcao = (20, 0)
```

Fonte: Autoria própria.

Explicação do código:

A detecção dos eventos de fechamento da janela é realizada nas (Linhas 27-29). A captura das teclas pressionadas pelo usuário e a alteração da direção de movimento da cobra acontecem nas (Linhas 31-39). Para o controle da movimentação, são utilizadas as setas direcionais do teclado: *UP*, *DOWN*, *LEFT* e *RIGHT*.

Figura 3.11: Resultado: Controle da cobra via teclado.



(a) UP (b) DOWN

(c) LEFT (d) RIGHT

Fonte: Autoria própria

Movimentação

A cobra se move continuamente na direção definida. A cada frame, calculamos a nova posição da cabeça com base na direção atual.

Código-fonte 68 Movimento da cobra.

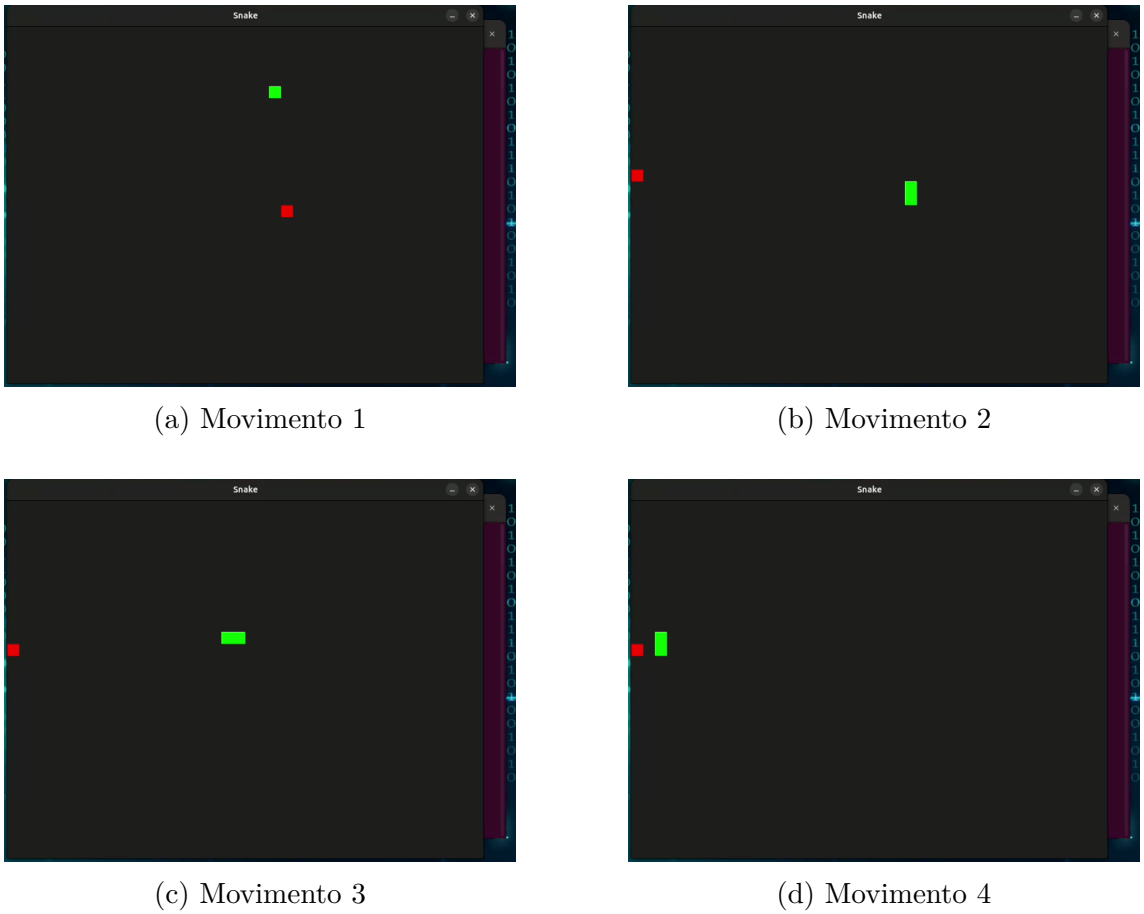
```
41      head_x = snake[0][0] + direcao[0]
42      head_y = snake[0][1] + direcao[1]
43
44      nova_head = (head_x, head_y)
```

Fonte: Autoria própria.

Explicação do código:

O cálculo da nova posição da cabeça da cobra, realizado a partir da soma da direção atual de movimento, acontece nas (Linhas 41-42). Já a criação da tupla contendo as novas coordenadas da cabeça é feita na (Linha 44).

Figura 3.12: Resultado: Movimento contínuo da cobra.



(a) Movimento 1 (b) Movimento 2

(c) Movimento 3 (d) Movimento 4

Fonte: Autoria própria

Colisões

O jogo deve detectar quando a cobra colide com as paredes da tela ou com o próprio corpo, encerrando o jogo nessas situações.

Código-fonte 69 Detecção de colisões.

```
46      if (
47          head_x < 0 or head_x >= LARGURA or
48          head_y < 0 or head_y >= ALTURA
49      ):
50          rodando = False
51
52      if nova_head in snake:
53          rodando = False
```

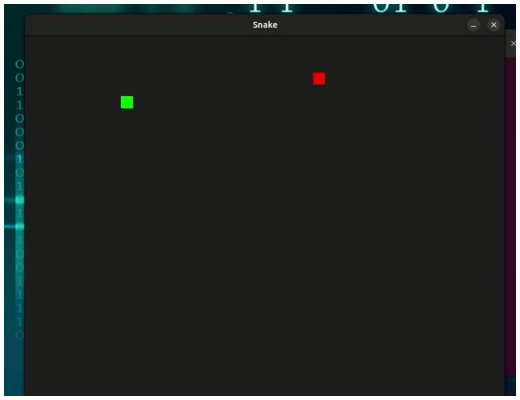
Fonte: Autoria própria.

Explicação do código:

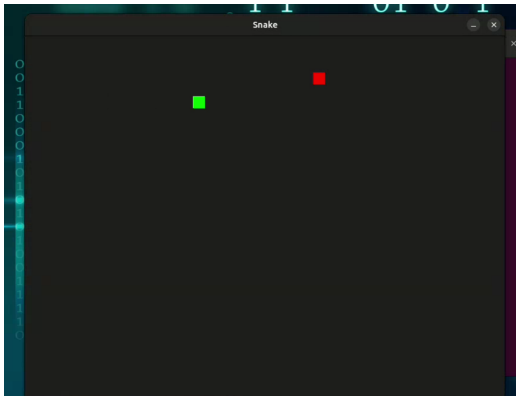
A verificação de colisão da cabeça da cobra com os limites da tela, identificando quando ela ultrapassa as bordas da janela, é realizada nas (Linhas 47-50). Já a detecção de colisão da cabeça com algum segmento do próprio corpo acontece nas

(Linhas 52-53).

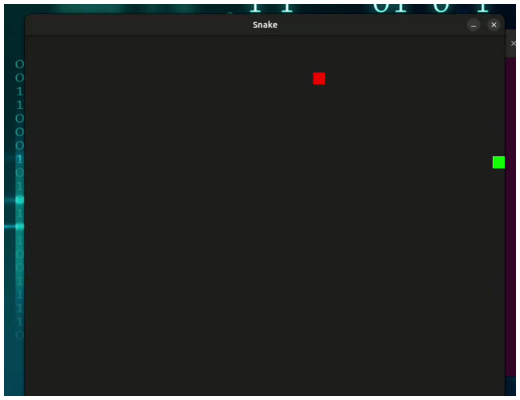
Figura 3.13: Resultado: Detecção de colisões encerra o jogo.



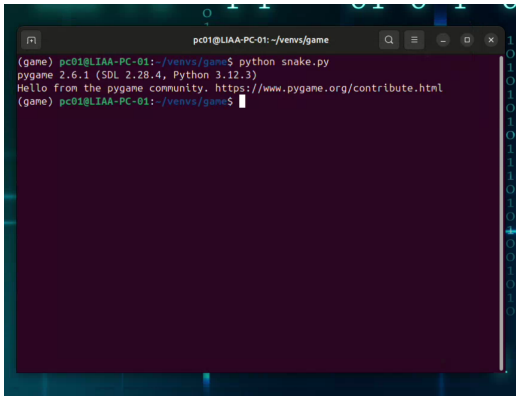
(a) Em movimento



(b) Em movimento



(c) Em movimento



(d) Encerramento após colisão

Fonte: Autoria própria

Crescimento

Quando a cobra come a comida, ela cresce. Caso contrário, ela apenas se move, removendo o último segmento da cauda.

Código-fonte 70 Crescimento da cobra.

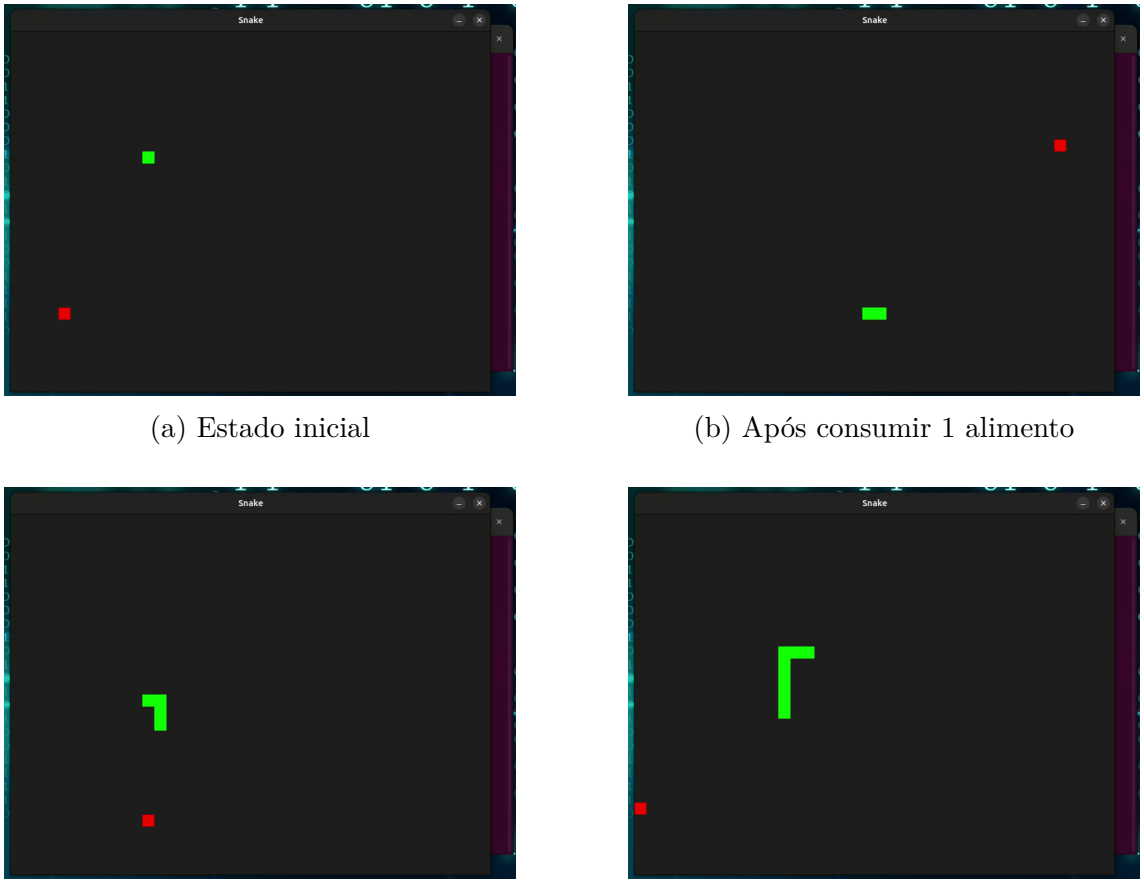
```
55      snake.insert(0, nova_head)
56
57      if nova_head == food_pos:
58          food_pos = (
59              random.randint(0, LARGURA // 20) * 20,
60              random.randint(0, ALTURA // 20) * 20
61          )
62      else:
63          snake.pop()
```

Fonte: Autoria própria.

Explicação do código:

A inserção da nova cabeça da cobra no início da lista que representa seu corpo é realizada na (Linha 55). A verificação de colisão da cobra com a comida e a geração de uma nova posição aleatória para o alimento acontecem nas (Linhas 57-61). Caso a cobra não consuma a comida, o último segmento do corpo é removido para manter o movimento contínuo, procedimento implementado nas (Linhas 62-63).

Figura 3.14: Resultado: Cobra crescendo ao consumir comida.



(a) Estado inicial

(b) Após consumir 1 alimento

(c) Crescimento contínuo

(d) Maior comprimento atingido

Fonte: Autoria própria

Renderização

Finalmente, renderizamos todos os elementos na tela: limpamos a tela, desenhamos a cobra e a comida, e atualizamos a exibição.

Código-fonte 71 Renderização.

```
65      tela.fill((30,30,30))
66
67      for segmento in snake:
68          pygame.draw.rect(tela, (0,255,0), (segmento[0], segmento[1], 20,
            ↪  20))
69
70      pygame.draw.rect(tela, (255,0,0), (food_pos[0], food_pos[1], 20, 20))
71
72      pygame.display.flip()
73
```

Fonte: Autoria própria.

Explicação do código:

A limpeza da tela utilizando uma cor de fundo cinza escuro é realizada na (Linha 65). O desenho de cada segmento da cobra na cor verde acontece nas (Linhas 67-68). Já a renderização da comida na cor vermelha é feita na (Linha 70). A atualização da tela do jogo, permitindo exibir todas as alterações gráficas, ocorre na (Linha 72). Por fim, o encerramento do Pygame ao término do loop principal é executado na (Linha 73).

Figura 3.15: Resultado: Estado final do jogo Snake em execução.

Autoria própria.

3.9 Declaração de uso de IA generativa

☐ NÃO utilizei ferramentas de IA generativa para produzir, reescrever ou expandir trechos do texto do meu mini-curso. Eventuais correções automáticas de ortografia/gramática do editor de texto não se enquadram nesta declaração.

☑ UTILIZEI ferramentas de IA generativa de forma ética e transparente, exclusivamente como apoio redacional (ex.: revisão gramatical, melhoria de clareza, sugestão de organização), sem terceirização do conteúdo intelectual. Declaro que: (a) informei o uso; (b) revisei integralmente e sou responsável(a) pelo texto final; (c) conferi a veracidade de dados, citações e referências, garantindo que não há referências inventadas, trechos sem fonte ou conteúdo não verificável.

Se marcou acima a opção UTILIZEI, descreva de forma objetiva:

- **Ferramenta(s)/versão(ões):** Utilizamos o ChatGPT (OpenAI), modelo GPT-5.3 (Instant), como apoio durante o desenvolvimento do mini curso;
- **Seções/trechos impactados:** A ferramenta foi usada apenas para auxiliar na organização das ideias e na estruturação geral do mini curso. Não houve geração de conteúdo final ou escrita de partes do trabalho;
- **Finalidade(s):** Usamos a IA como apoio para sugerir formas de organizar melhor o conteúdo e melhorar a clareza das ideias durante o planejamento do mini curso. Todo o conteúdo final foi desenvolvido, revisado e validado pelo grupo, sem terceirização da produção intelectual.

3.10 Bibliografia

Atomic Heritage Foundation (2026). William “willy” higinbotham. Disponível em: https://ahf.nuclearmuseum.org/ahf/profile/william-willy-higinbotham/. Acesso em: 01 maio 2026.

Brookhaven National Laboratory (1958). Tennis for two: The first video game. Disponível em: https://www.bnl.gov/about/history/firstvideo.php. Acesso em: 01 maio 2026.

Computer History Museum (1962). Pdp-1 and spacewar. Disponível em: https://www.computerhistory.org/pdp-1/. Acesso em: 01 maio 2026.

Foundation, P. S. (2026). venv — criação de ambientes virtuais. https://docs.python.org/pt-br/3/library/venv.html. Acesso em: 17 maio 2026.

Pygame Community (2025). Pygame documentation. *Pygame Official Documentation*. Documentação oficial da biblioteca Pygame.

Wikipedia Contributors (2026). Steve russell (computer scientist). Disponível em: https://en.wikipedia.org/wiki/Steve_Russell_%28computer_scientist%29. Acesso em: 01 maio 2026.

Minicurso 4

Vue.js na prática, do zero ao primeiro projeto

Stephanny Kauane Oliveira Amaral e Eduardo Augusto Oliveira Goular

Resumo

Este trabalho apresenta a concepção de um minicurso prático voltado ao desenvolvimento de interfaces web com o framework Vue.js e a ferramenta de build Vite. A proposta guia o aluno de forma incremental desde a configuração do ambiente (Node.js, NPM e VS Code) até a construção de uma aplicação funcional de gerenciamento de tarefas. São explorados conceitos essenciais como reatividade, manipulação automatizada do DOM, diretivas estruturais e de ligação (v-model, v-if, v-for), além da arquitetura de componentes baseada em blocos únicos (.vue). Por fim, aborda-se a comunicação entre componentes via props e eventos ($emit), consolidando o uso do Vue.js para a criação de Single Page Applications (SPAs) modernas, escaláveis e de rápida curva de aprendizado.

4.1 Introdução

Este mini curso tem como objetivo ensinar o desenvolvimento de interfaces utilizando Vue.js de forma prática e acessível. O conteúdo foi cuidadosamente estruturado para que iniciantes consigam acompanhar desde a instalação do ambiente até a entrega de uma aplicação funcional, sem lacunas no aprendizado.

Ao longo do material, você avançará de forma progressiva: primeiro entenderá **o que é** o Vue.js e por que ele existe, depois aprenderá seus **fundamentos** com exemplos comentados, e por fim colocará tudo em prática com um **projeto real**.

Dica

Como usar este material: leia cada seção na ordem apresentada. Não pule etapas, cada conceito é construído sobre o anterior. Sempre que aparecer um

bloco de código, tente digitá-lo (não copiar) no seu editor. Isso ajuda a fixar a sintaxe.

O que você vai aprender

- O que é o Vue.js e por que usá-lo.
- O que é o Vite e qual o seu papel no projeto.
- Como configurar o ambiente de desenvolvimento passo a passo.
- A estrutura de um componente Vue (`template`, `script`, `style`).
- As principais diretivas: `v-model`, `v-if`, `v-for`, `v-bind`, `v-show`.
- Como capturar e tratar eventos do usuário.
- Como desenvolver uma lista de tarefas completa.
- Como criar componentes reutilizáveis com *props* e *emit*.

Dica

Não é necessário ter experiência prévia com frameworks. Conhecimento básico de HTML e JavaScript é suficiente para acompanhar este curso.

Figura 4.1: Mapa do curso.

Fonte: Autoria própria.

4.2 O que é o Vue.js?

O Vue.js é um **framework JavaScript progressivo** utilizado para a construção de interfaces de usuário (UI), especialmente em aplicações web.

Ele foi criado com o objetivo de ser simples, flexível e fácil de integrar, permitindo que desenvolvedores construam desde pequenas interações em páginas já existentes até aplicações completas, as chamadas *Single Page Applications* (SPAs).

A palavra **progressivo** é importante: você pode adotar o Vue aos poucos no seu projeto, adicionando funcionalidades conforme a necessidade, sem precisar reescrever tudo de uma vez.

Por que usar Vue.js?

- **Fácil de aprender:** curva de aprendizado suave, ideal para quem está começando.
- **Organização:** utiliza componentes reutilizáveis que facilitam a manutenção do código.
- **Reatividade:** a interface se atualiza automaticamente quando os dados mudam.
- **Leve e performático:** ótimo desempenho mesmo em aplicações maiores.
- **Grande ecossistema:** conta com ferramentas como Vue Router (navegação entre páginas) e Pinia (gerenciamento de estado).

Onde o Vue é utilizado?

Antes de entrar nos detalhes técnicos, vale entender em quais contextos o Vue aparece no mundo real:

- Sistemas web completos (ERPs, portais corporativos)
- Dashboards administrativos
- Aplicações interativas (e-commerce, fóruns, redes sociais)
- Integração com projetos legados já existentes

Figura 4.2: Comparações entre *frameworks*.

Fonte: Autoria própria.

O problema: como era antes do Vue?

Para entender o valor do Vue, é preciso relembrar como as interfaces eram construídas antes dos frameworks modernos: manipulando o DOM diretamente com JavaScript puro.

Conceito importante

O que é o DOM?

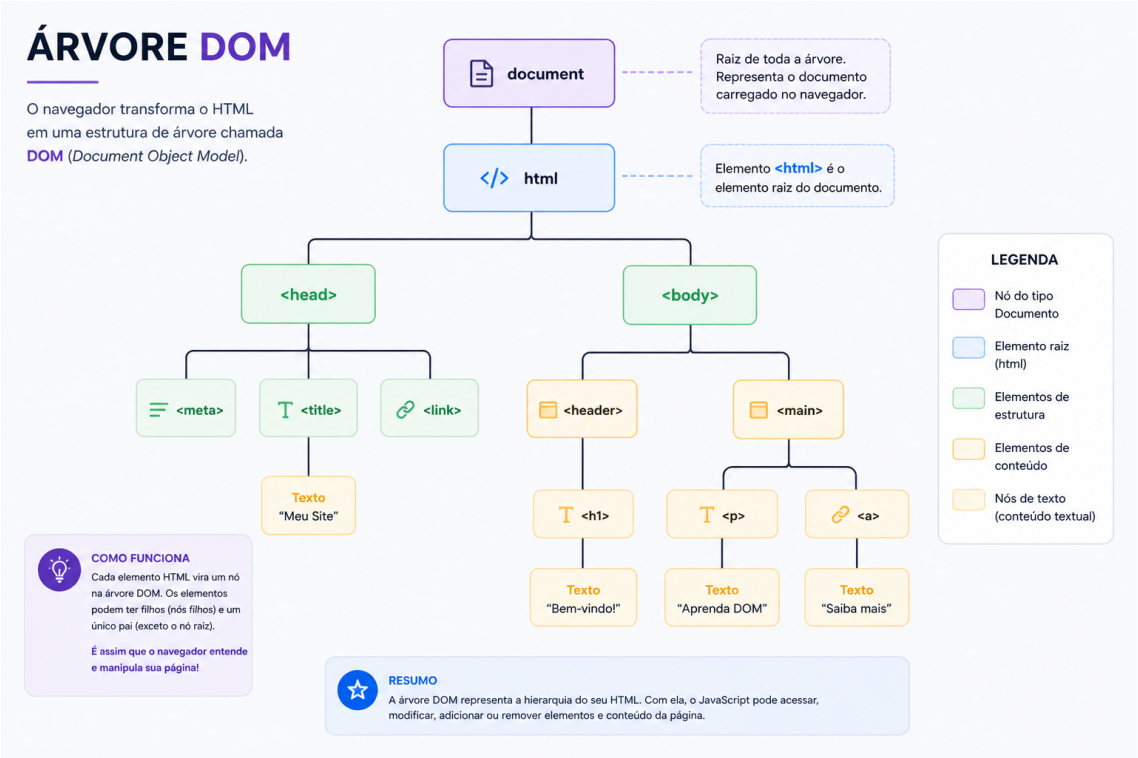
DOM significa *Document Object Model*. É a representação em árvore de todos os elementos de uma página HTML. Quando o navegador carrega uma página, ele cria essa estrutura para que o JavaScript consiga acessar e modificar os elementos.

Exemplo simplificado de uma árvore DOM:

```
document
|__ html
|__ head
|   |__ title -> "Minha Página"
|__ body
|__ h1 -> "Olá, Mundo!"
|__ p  -> "Bem-vindo ao Vue."
```

Cada item nessa árvore é um **nó**. O JavaScript puro permite que você acesse e altere esses nós manualmente.

Figura 4.3: Árvore DOM.



Fonte: Autoria própria.

Sem um framework, para atualizar o texto de um título na tela, você precisaria escrever algo assim:

```
// Acessando um elemento pelo ID e alterando seu texto
document.getElementById("titulo").innerText = "Novo texto";

// Para criar um item de lista dinamicamente:
const li = document.createElement("li");
li.textContent = "Nova tarefa";
document.getElementById("lista").appendChild(li);
```

Dica

Repare que no código acima você precisa **dizer ao navegador exatamente o que fazer**: “busque esse elemento pelo ID”, “crie esse elemento”, “adicione-o ali”. Com Vue, você não faz mais isso, você apenas muda os dados e o framework cuida do resto.

Isso funciona em projetos simples, mas traz problemas sérios à medida que o projeto cresce:

- O código se torna difícil de ler e manter.
- É fácil cometer erros ao misturar lógica de negócio com manipulação de interface.
- Qualquer mudança na estrutura HTML pode quebrar silenciosamente o JavaScript.

A solução: como o Vue resolve esse problema

Com o Vue, você não precisa mais dizer ao navegador *como* atualizar a tela: você apenas atualiza os dados, e o Vue cuida do resto.

O segredo por trás disso é um conceito chamado **reatividade**. Pense da seguinte forma: os dados são como uma torneira, e a interface é o copo. Sempre que a torneira é aberta (dado muda), o copo é preenchido automaticamente (tela atualiza). Você não precisa carregar água manualmente.

O fluxo na prática é:

1. O usuário interage com a interface (clica em um botão, digita algo).
2. O Vue detecta a mudança nos dados.
3. A interface é atualizada automaticamente no `<template>`.

Você não manipula o HTML diretamente. Você **manipula os dados**, e o Vue faz o resto.

```
// Com Vue, basta mudar o dado:
this.titulo = "Novo texto";
// A interface é atualizada automaticamente.
```

Figura 4.4: Reatividade do Vue.

Fonte: Autoria própria.

4.3 Fundamentos do Vue.js

Agora que você entende *por que* o Vue existe, é hora de colocá-lo em funcionamento. Nesta seção você vai configurar o ambiente, criar seu primeiro projeto e aprender as ferramentas fundamentais da linguagem.

Dica

Onde estamos? Esta seção é o coração teórico do curso. Leia com calma, execute os exemplos no seu editor e não avance para o projeto prático (Seção 4) sem ter entendido os conceitos aqui apresentados.

Configurando o ambiente e criando o projeto

Conceito importante

O que é o Vite?

Ao criar um projeto Vue moderno, usamos o **Vite** (pronuncia-se "vit", do francês *rápido*) como ferramenta de build. Mas o que isso significa?

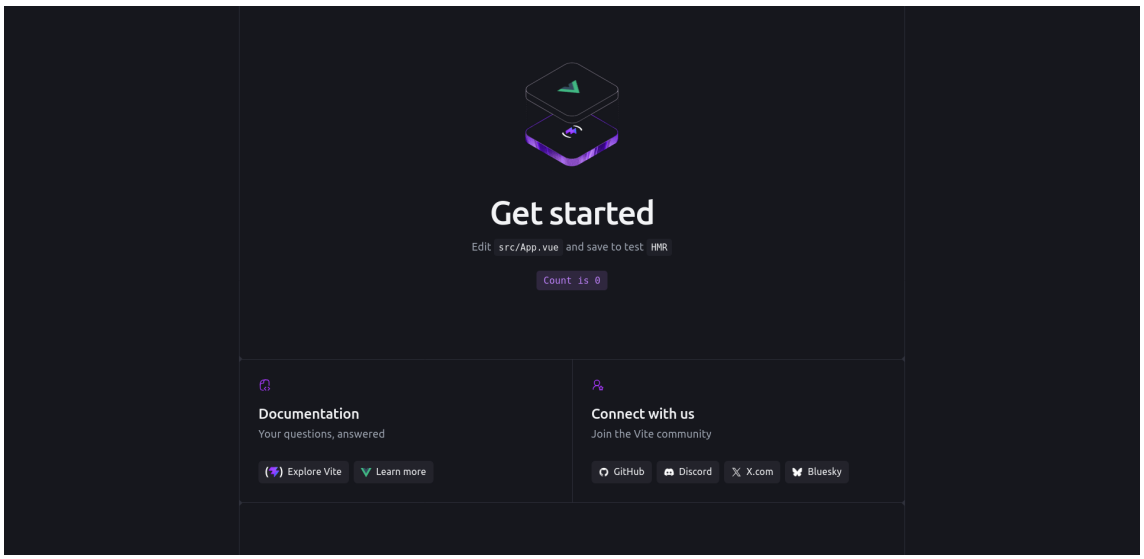
Quando você escreve código Vue, o navegador não entende diretamente os arquivos `.vue`, ele só entende HTML, CSS e JavaScript puro. O Vite é o responsável por **transformar** esses arquivos em algo que o navegador consegue processar.

Além disso, o Vite oferece:

- **Servidor de desenvolvimento instantâneo:** ao salvar um arquivo, a página atualiza em milissegundos.
- **Hot Module Reload (HMR):** apenas o componente alterado é recarregado, sem perder o estado da aplicação.
- **Build otimizado:** ao gerar a versão final do projeto, o Vite comprime e otimiza os arquivos automaticamente.

Em resumo: o **Vue** é o framework que você usa para escrever a interface, e o **Vite** é a ferramenta que torna o desenvolvimento com Vue rápido e agradável. Os dois trabalham juntos.

Figura 4.5: Tela incial Vue + Vite.



Fonte: Autoria própria.

Ferramentas necessárias

Antes de começar, você precisará instalar as ferramentas abaixo. Cada uma tem um papel específico:

- **Node.js:** ambiente que permite executar JavaScript fora do navegador. É obrigatório para rodar o Vue e suas ferramentas. Pense nele como o "motor" que alimenta tudo.
- **NPM:** gerenciador de pacotes que vem junto com o Node.js, usado para instalar dependências. É como uma "loja de bibliotecas" para o seu projeto.
- **Visual Studio Code:** editor de código leve, gratuito e com excelentes extensões para Vue.
- **Navegador web:** Chrome, Firefox ou outro de preferência. Você usará para visualizar sua aplicação.

Instalando o Node.js

1. Acesse o site oficial do Node.js Node.js [2026].
2. Escolha a versão **LTS** (Long Term Support, a mais estável).
3. Baixe e instale normalmente para o seu sistema operacional.

Figura 4.6: Download do Node.js versão LTS.

Fonte: Autoria própria.

Após a instalação, abra o terminal (no Windows: `cmd` ou `PowerShell`; no Mac/Linux: `Terminal`) e verifique se tudo correu bem digitando:

```
$ node -v
v20.11.0
$ npm -v
10.2.4
```

Se aparecerem os números de versão (os valores exatos podem ser diferentes), a instalação foi bem-sucedida. Se aparecer uma mensagem de erro como `command not found`, feche e reabra o terminal, ou repita a instalação.

Instalando o Visual Studio Code

1. Acesse o site oficial do VS Code: Visual Studio Code [2026].
2. Clique em **Download** e escolha a versão correspondente ao seu sistema operacional (Windows, macOS ou Linux).
3. Execute o instalador e siga as instruções. No Windows, recomenda-se marcar a opção **"Add to PATH"** durante a instalação, isso permite abrir o VS Code diretamente pelo terminal.
4. Após a instalação, abra o VS Code e verifique se ele abre corretamente.

Dica

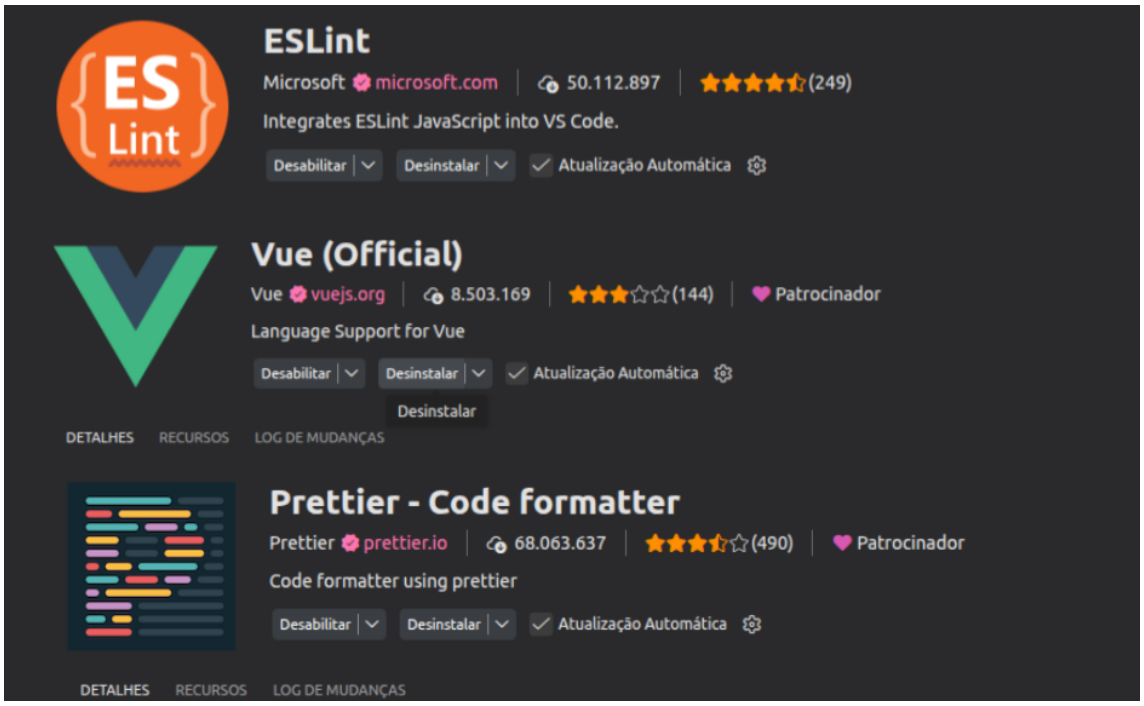
Para abrir uma pasta de projeto diretamente pelo terminal, use o comando `code .` dentro da pasta desejada. Isso só funciona se a opção **"Add to PATH"** tiver sido marcada durante a instalação.

Extensões recomendadas para o VS Code

Após abrir o VS Code, instale as extensões a seguir. Para instalar, clique no ícone de extensões na barra lateral esquerda (ou pressione `Ctrl+Shift+X`) e pesquise pelo nome:

- **Vue - Official** — suporte completo ao Vue.js (realce de sintaxe, autocomplete). **Esta é a mais importante.**
- **Prettier** — formatação automática de código ao salvar.
- **ESLint** — detecta erros no código em tempo real, antes mesmo de você rodar o projeto.

Figura 4.7: Extensões para o VS Code.



Fonte: Autoria própria.

Criando o projeto com Vite

Agora vamos criar o projeto. Abra o terminal, escolha uma pasta onde deseja salvar seus projetos e execute os comandos abaixo **um por vez**:

```
# Criar pasta de projetos e entrar nela
mkdir projetos-vue
cd projetos-vue

# Criar o projeto com Vite
npm create vite@latest
```

O assistente interativo pedirá algumas informações. Use as setas do teclado para navegar e `Enter` para confirmar. Preencha conforme abaixo:

```
> Project name:  meu-projeto (o nome de sua preferência)
> Select a framework:  Vue
> Select a variant:  JavaScript
```

Atenção

Escolha **JavaScript** (não TypeScript) neste momento. TypeScript adiciona complexidade desnecessária para quem está aprendendo o Vue pela primeira

vez.

Em seguida, o Vite criará uma pasta com o nome do seu projeto. Agora acesse essa pasta, instale as dependências e inicie o servidor de desenvolvimento:

```
cd meu-projeto
npm install
npm run dev
```

Dica

O comando `npm install` baixa todas as bibliotecas que o projeto precisa. Isso pode demorar alguns segundos. O comando `npm run dev` inicia o servidor local. Você só precisa rodá-lo uma vez por sessão de trabalho.

Se tudo correu bem, você verá no terminal algo parecido com:

```
VITE v5.x.x  ready in 300 ms

Local:   http://localhost:5173/
Network: use --host to expose
```

Abra o navegador e acesse `http://localhost:5173/`. Você verá a página inicial padrão do Vue com Vite e isso significa que o projeto está funcionando!

Figura 4.8: Integração Vue e Vite.

Fonte: Autoria própria.

Estrutura de pastas do projeto

Com o projeto criado, o Vite gera automaticamente a seguinte estrutura de arquivos. Veja o que cada pasta e arquivo significa:

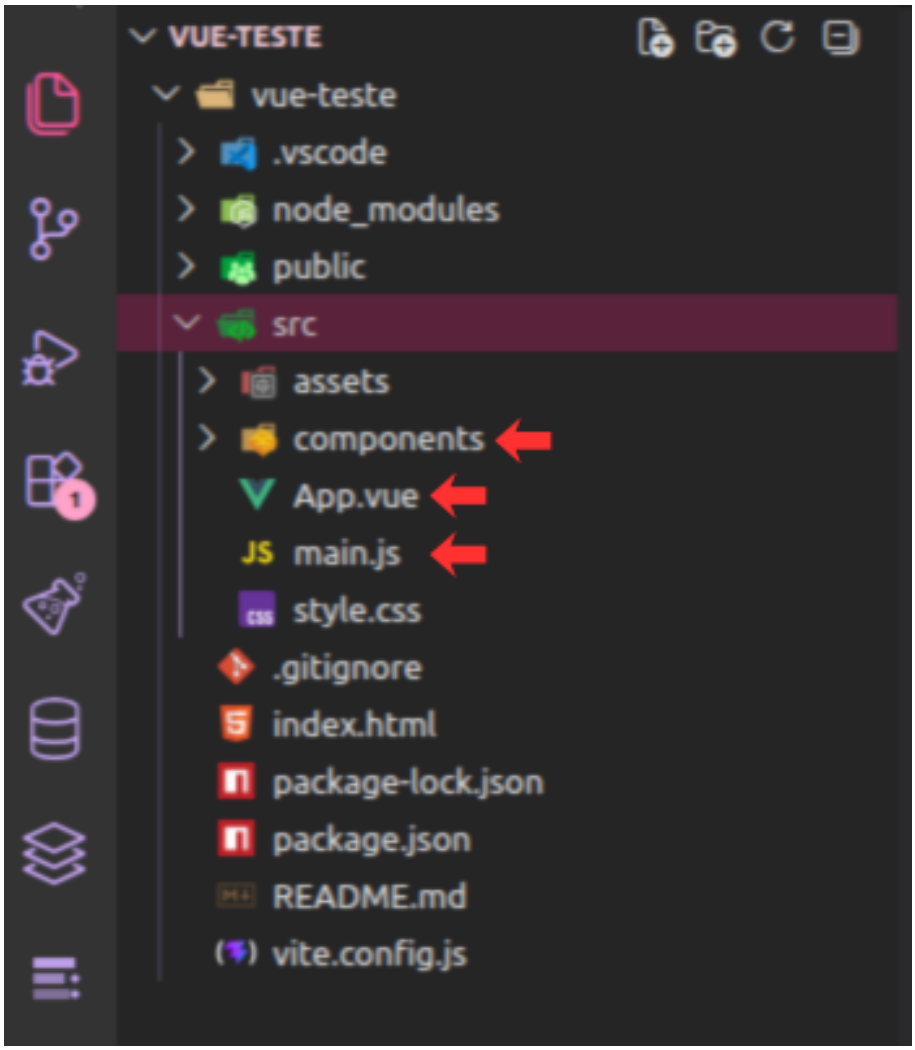
Os arquivos mais importantes para você agora são:

- **`src/App.vue`** — é o componente principal, o ponto de partida visual da aplicação. É aqui que você começará a escrever.
- **`src/main.js`** — inicializa o Vue e "monta" a aplicação dentro do `index.html`. Você raramente precisará alterar este arquivo.
- **`src/components/`** — pasta onde você criará seus próprios componentes reutilizáveis ao longo do curso.

Dica

A pasta `node_modules/` pode ter centenas de megabytes e nunca deve ser enviada ao GitHub. O Vite já configura o arquivo `.gitignore` automaticamente para ignorá-la. Se você clonar um projeto sem essa pasta, basta rodar `npm install` para recriá-la.

Figura 4.9: Estrutura de pastas do Vue.



Fonte: Autoria própria.

Estrutura de um componente Vue

Agora que o projeto existe e você conhece sua estrutura de pastas, é hora de entender o arquivo mais importante: o componente `.vue`.

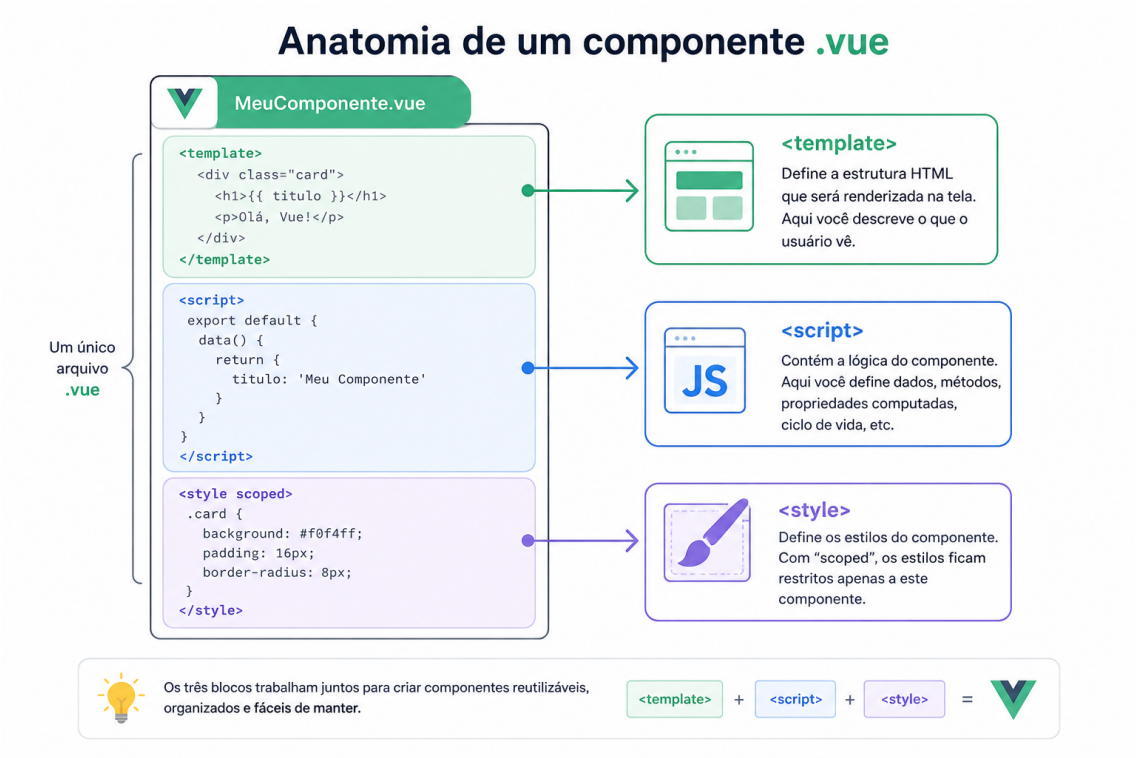
No Vue.js, a principal unidade de construção é o **componente**. Um componente é um arquivo com extensão `.vue` que reúne, em um único lugar, a estrutura, a lógica e o estilo de uma parte da interface.

Todo componente Vue é dividido em três blocos obrigatórios (na prática, o `<style>` é opcional, mas boa prática inclui sempre):

- **`<template>`** → o que será exibido na tela (HTML). Tudo que o usuário vê vem daqui.
- **`<script>`** → a lógica do componente: dados, funções, comportamento.

- **<style>** → a aparência visual (CSS). Pode ser scoped para não afetar outros componentes.

Figura 4.10: Anatomia do componente .vue.



Fonte: Autoria própria.

```
1   <template>
2   <h1 class="titulo">{{ mensagem }}</h1>
3   </template>
4
5   <script>
6   export default {
7       data() {
8           return {
9               mensagem: "Olá, Mundo!"
10          }
11      }
12  }
13  </script>
14
15  <style>
16  .titulo {
17      color: #42b883; /* verde Vue */
18      text-align: center;
19  }
```

```
20          </style>
```

Entendendo o exemplo passo a passo:

- `{{ mensagem }}` é chamado de **interpolação**. O Vue substitui esse marcador pelo valor atual da variável `mensagem`. Se `mensagem` for `"Olá, Mundo!"`, o resultado no HTML será `<h1>Olá, Mundo!</h1>`. Você nunca precisa tocar no DOM.
- `data()` é uma função que retorna um objeto com todas as variáveis reativas do componente. Qualquer mudança nessas variáveis atualiza a tela automaticamente. **Todas as variáveis que você quiser usar no template devem ser declaradas aqui.**
- `<style>` define o CSS aplicado ao componente. Sem o atributo `scoped`, os estilos valem para a página toda; com ele, valem apenas para este componente.

Dica

O `<style>` pode receber o atributo `scoped`, assim: `<style scoped>`. Isso garante que os estilos daquele componente não vazem para outros, evitando conflitos de CSS. Use `scoped` sempre que possível.

Atenção

Experimente agora! Abra o arquivo `src/App.vue` no VS Code, apague o conteúdo e substitua pelo exemplo acima. Salve o arquivo (`Ctrl+S`) e olhe o navegador, você verá a atualização em tempo real!

Diretivas do Vue

Com a estrutura de um componente clara, podemos explorar as **diretivas**, os comandos especiais do Vue que usamos diretamente no HTML para adicionar comportamentos dinâmicos.

Todas as diretivas começam com o prefixo `v-`. Quando você vir `v-` em um atributo HTML, saiba que é o Vue quem vai interpretá-lo.

Conceito importante

Pense nas diretivas como instruções especiais que você dá ao Vue: "*repita este elemento para cada item da lista*", "*mostre este parágrafo somente se o usuário estiver logado*", "*conecte este campo ao dado nome*". O Vue interpreta essas instruções e age automaticamente.

Figura 4.11: Diretivas do Vue.

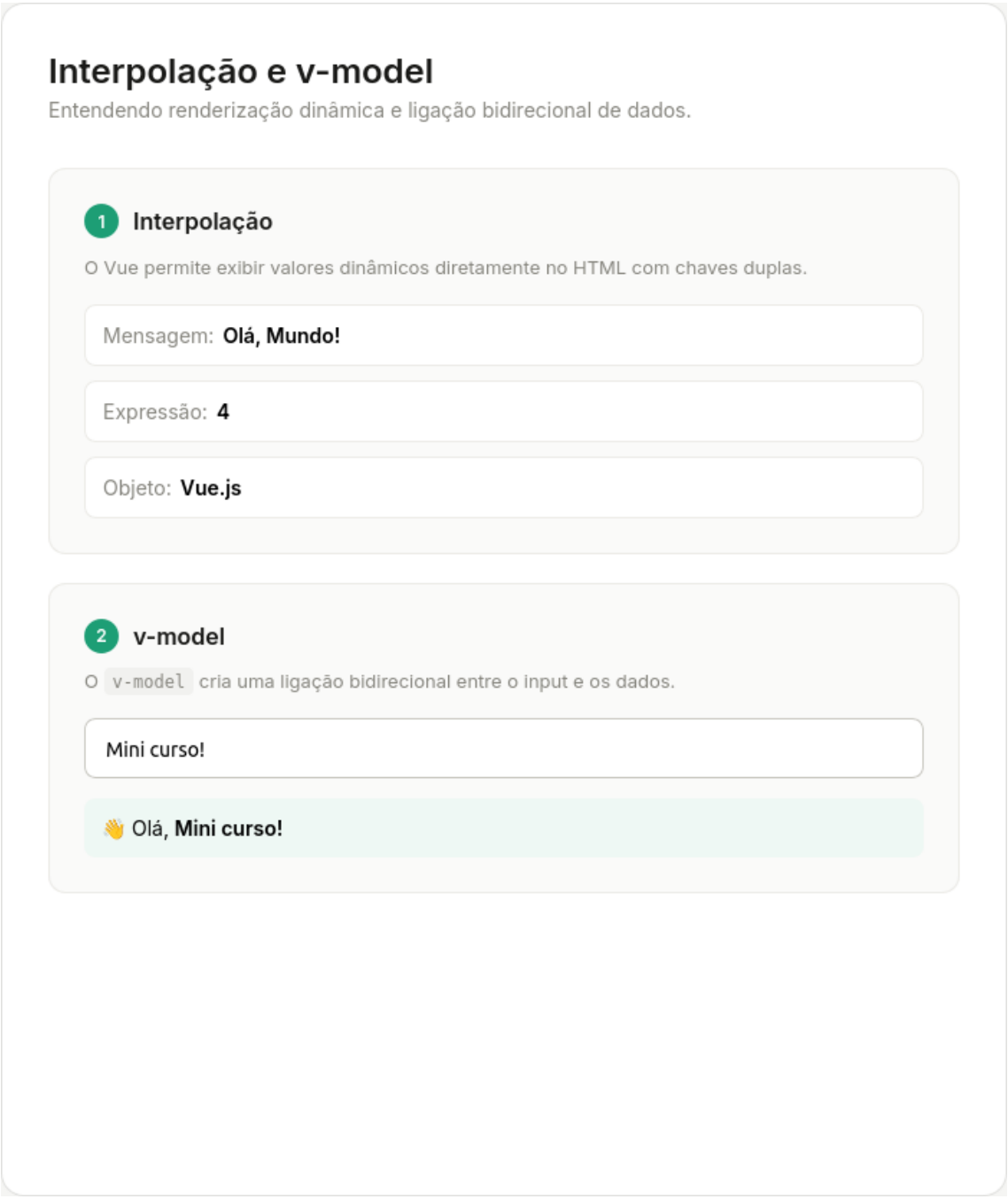
Fonte: Autoria própria.

Interpolação — {{ }}

A forma mais simples de exibir um dado na tela. Os valores entre {{ }} são avaliados como expressões JavaScript:

```
<p>{{ mensagem }}</p>
<!-- Se mensagem = "Olá mundo!", exibe: Olá mundo! -->

<p>{{ 2 + 2 }}</p>
<!-- Exibe: 4 -->

<p>{{ usuario.nome }}</p>
<!-- Exibe a propriedade nome do objeto usuario -->
```

Figura 4.12: Exemplo de interpolação exibida pelo código do mini curso.



Fonte: Autoria própria.

v-model — Ligação bidirecional

A diretiva `v-model` conecta um campo de entrada diretamente a uma variável. Qualquer coisa digitada pelo usuário é automaticamente salva na variável; se a variável mudar no código, o campo também é atualizado.

Dica

`v-model` é chamado de **data binding bidirecional**: os dados fluem do JavaScript para o HTML *e* do HTML de volta para o JavaScript. É como uma

via de mão dupla entre o campo e a variável.

```
        <template>
        <input v-model="nome" placeholder="Digite seu nome">
        <p>Olá, {{ nome }}!</p>
        </template>

        <script>
        export default {
                data() {
                        return {
                                nome: ""  <!-- variável ligada ao campo
↪  acima -->
                        }
                }
        }
        </script>
```

Teste você mesmo: cole esse código em `App.vue`, salve e comece a digitar no campo. Observe como o parágrafo abaixo é atualizado a cada letra digitada, isso é a reatividade do Vue em ação.

v-bind — Atributos dinâmicos

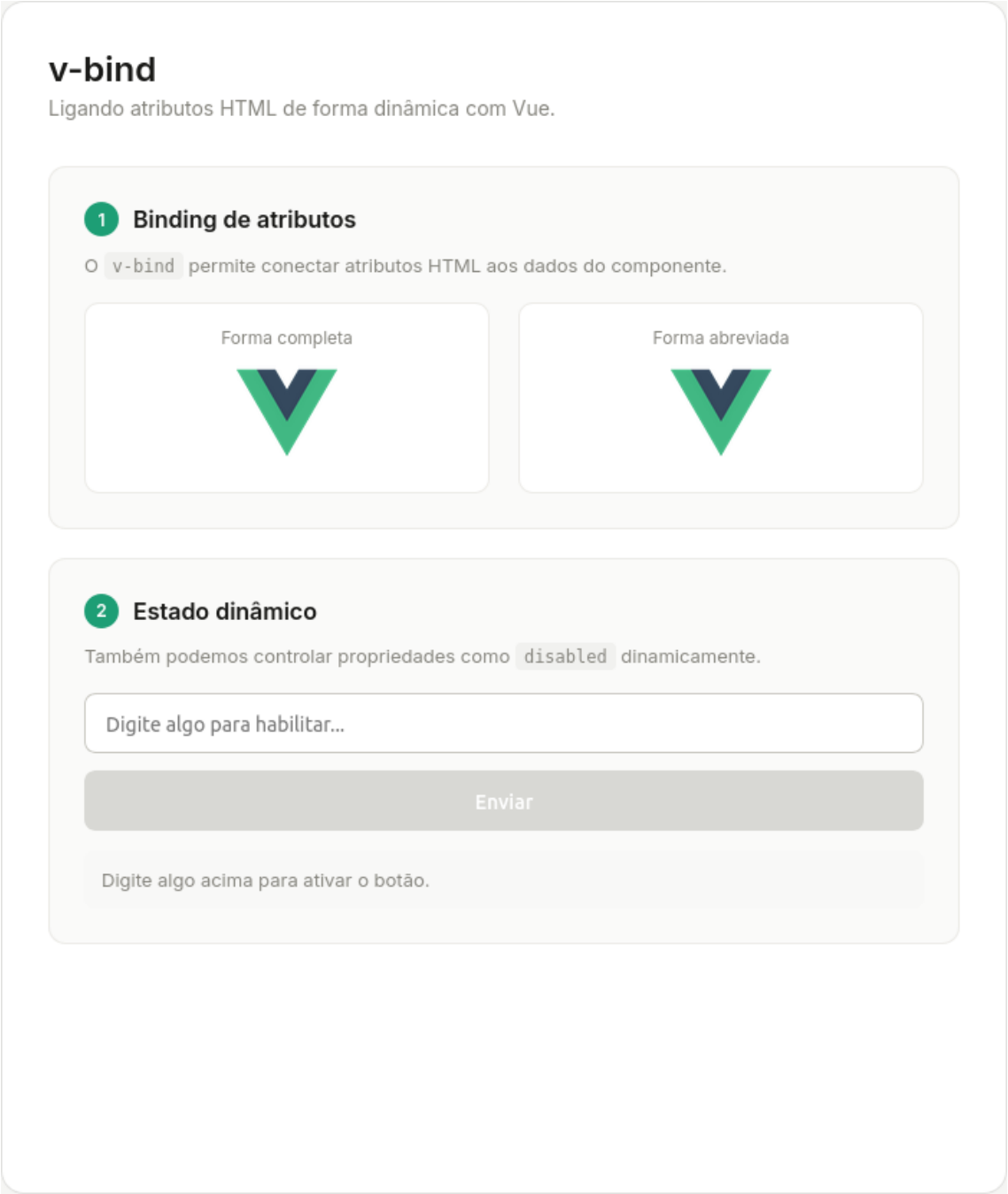
Vincula atributos HTML a valores dinâmicos. Sua forma abreviada é `:` (dois pontos). Use quando precisar que o valor de um atributo HTML venha de uma variável.

```
        <!-- Forma completa -->
        <img v-bind:src="urlDaImagem" v-bind:alt="descricao">

        <!-- Forma abreviada (preferida na prática) -->
        <img :src="urlDaImagem" :alt="descricao">

        <!-- Outro exemplo: desabilitar botão dinamicamente -->
        <button :disabled="formularioVazio">Enviar</button>
```

Dica

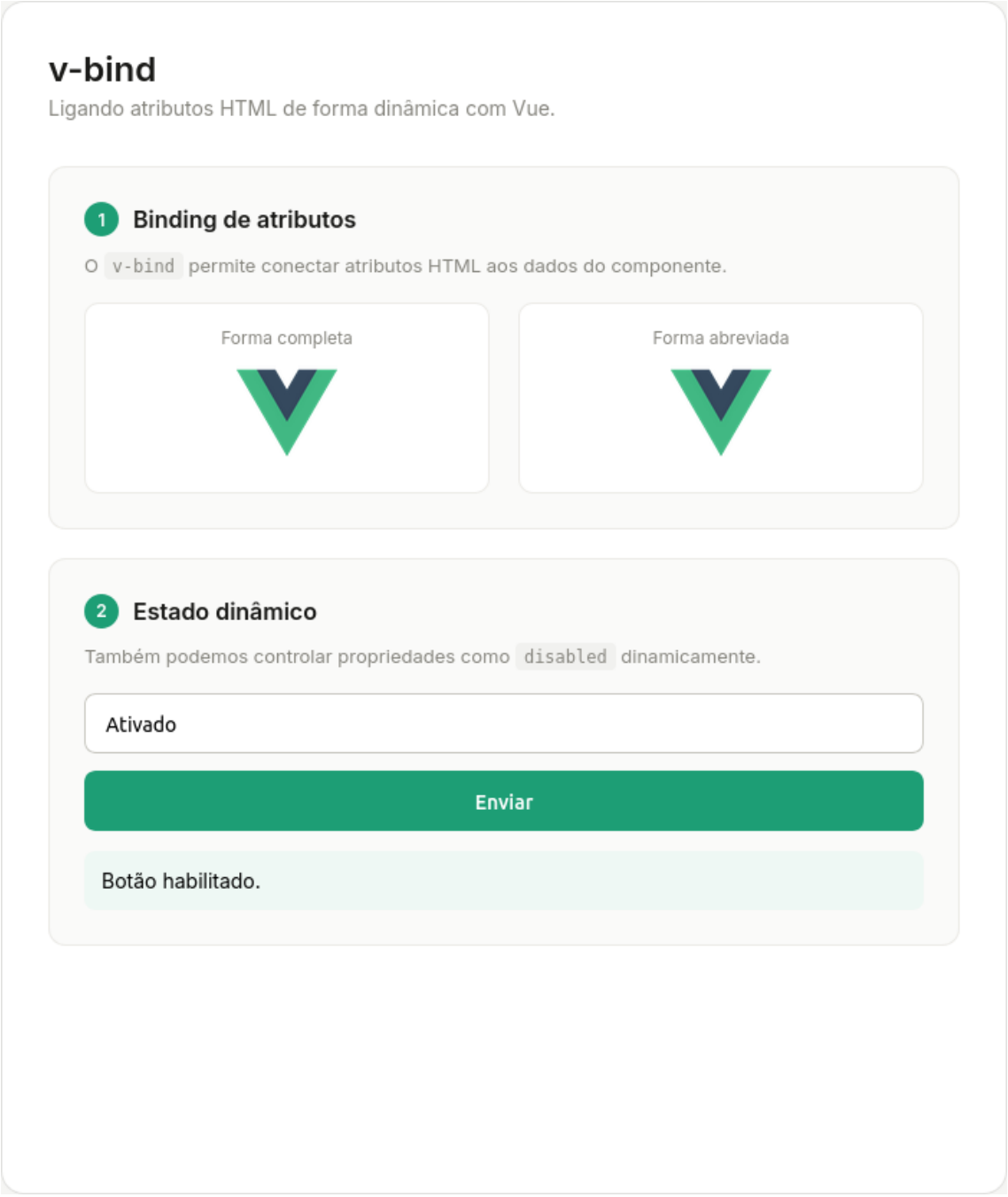
Na prática, quase todos os desenvolvedores Vue usam a forma abreviada `:` em vez de `v-bind:`. As duas são equivalentes.

Figura 4.13: Exemplo de v-bind exibida pelo código do mini curso.



Fonte: Autoria própria.

Figura 4.14: Comportamento do v-bind após uso da diretiva.



Fonte: Autoria própria.

v-if, v-else-if e v-else — Renderização condicional

Exibe ou **remove** elementos com base em condições. Funciona exatamente como um `if/else` do JavaScript, mas aplicado ao HTML.

```
1        <p v-if="nota >= 7">Aprovado</p>
2        <p v-else-if="nota >= 5">Recuperação</p>
3        <p v-else>Reprovado</p>
```

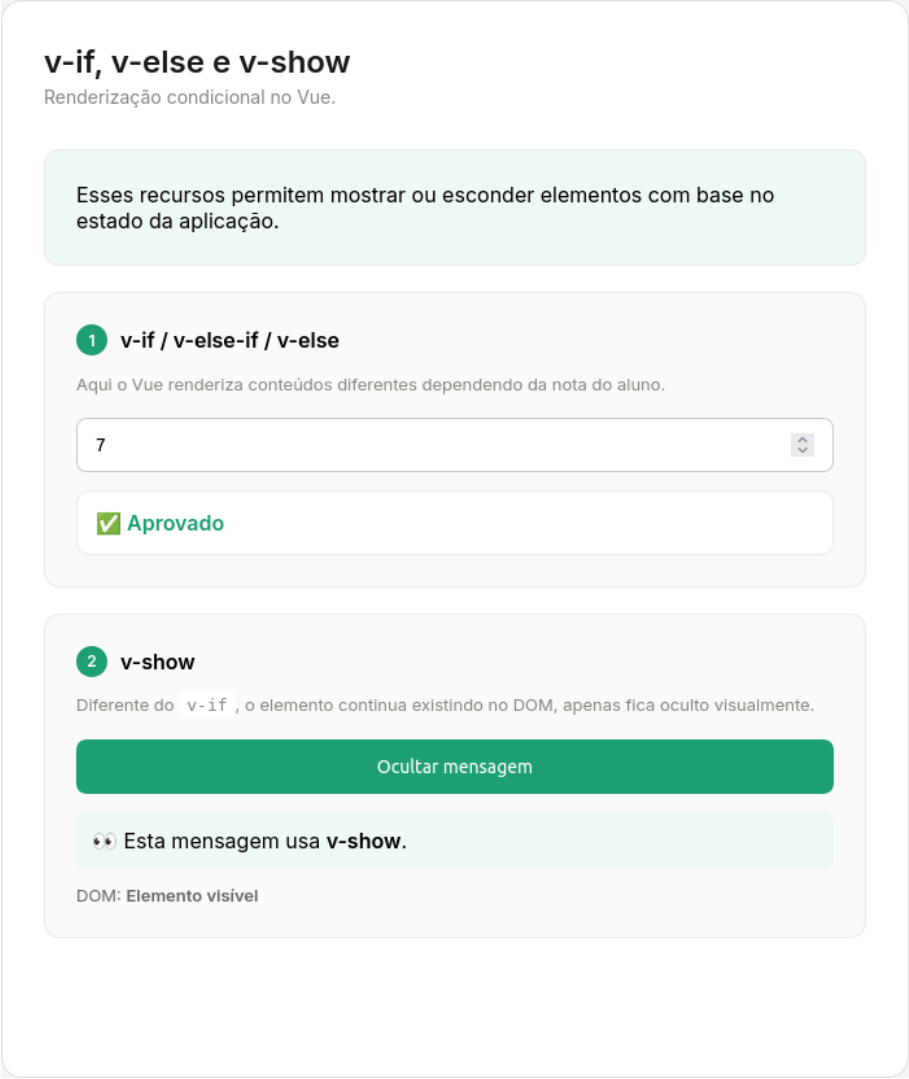
Conceito importante

v-if vs v-show — qual usar?

- `v-if` — **remove** o elemento completamente do DOM quando a condição é falsa. O elemento não existe na página, economizando memória. Recomendado para elementos que raramente mudam de visibilidade (ex.: mensagem de erro que aparece apenas uma vez).
- `v-show` — **oculta** o elemento com `display: none`, mas ele permanece no DOM. Mais eficiente para alternâncias frequentes (ex.: um menu que abre e fecha constantemente), pois evita o custo de criar/destruir o elemento repetidamente.

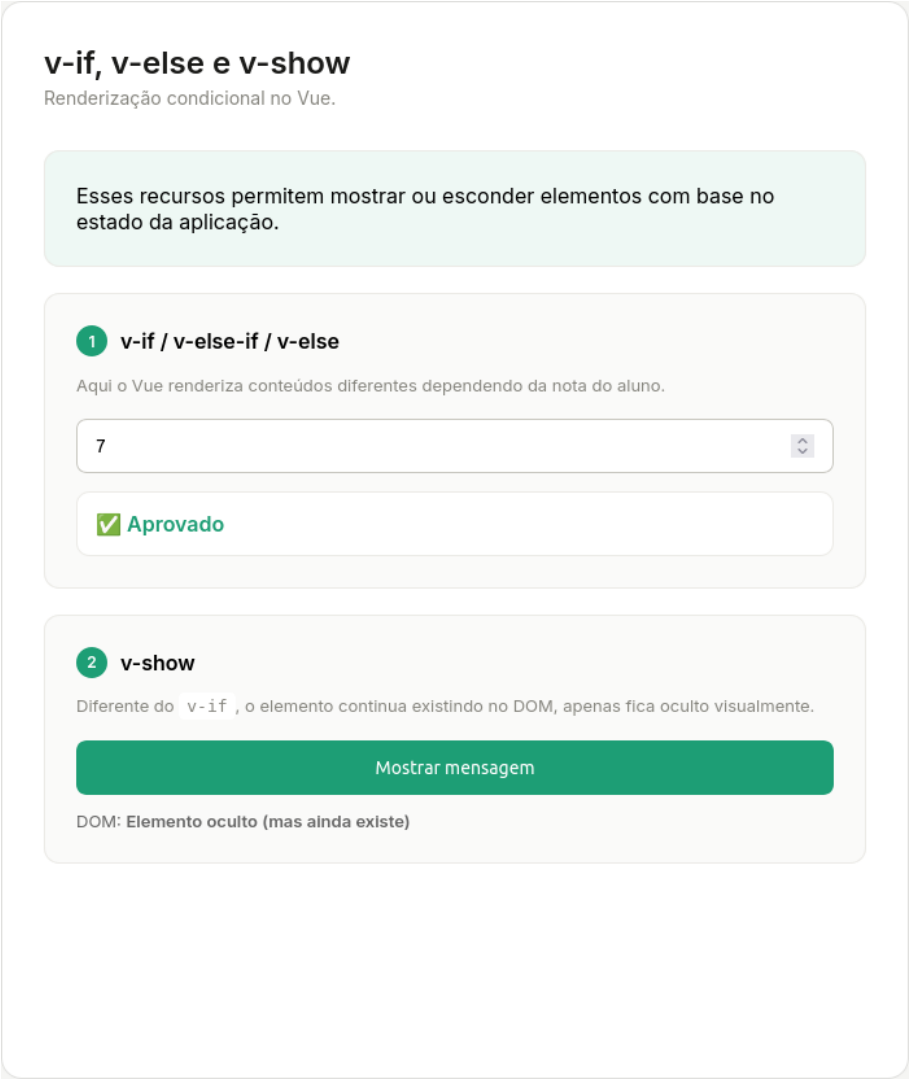
Regra prática: se o elemento alterna com frequência, use `v-show`. Se aparece raramente, use `v-if`.

Figura 4.15: Exemplo de renderização condicional exibida pelo código do mini curso.



Fonte: Autoria própria.

Figura 4.16: Comportamento do v-show após uso da diretiva.



Fonte: Autoria própria.

v-for — Renderização de listas

Itera sobre um array e cria um elemento para cada item. É o equivalente Vue de um loop `for` no template.

```
<ul>
<li v-for="(item, index) in frutas" :key="index">
{{ index + 1 }} - {{ item }}
</li>
</ul>
<!-- Se frutas = ["Maçã", "Banana", "Uva"], exibe:
1 - Maçã
2 - Banana
3 - Uva  -->
```

Dica

A sintaxe `(item, index) in lista` dá acesso ao **valor** (`item`) e à **posição** (`index`) de cada elemento. O `index` começa em 0, por isso usamos `index + 1` para exibir a numeração a partir de 1.

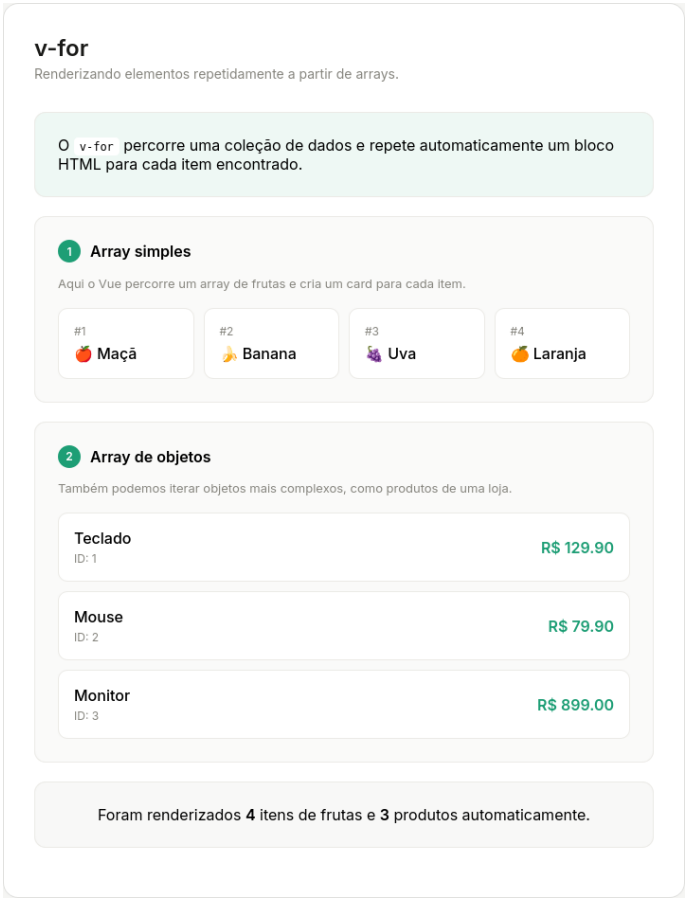
Atenção

Sempre forneça o atributo `:key` com um valor único para cada item do `v-for`. Sem ele, o Vue pode ter dificuldades para rastrear atualizações e causar comportamentos inesperados na renderização, especialmente ao adicionar ou remover itens da lista.

v-for com objetos: quando cada item da lista é um objeto, você acessa suas propriedades normalmente:

```
1        <li v-for="produto in produtos" :key="produto.id">
2        {{ produto.nome }} - R$ {{ produto.preco }}
3        </li>
```

Figura 4.17: Exemplo de v-for exibida pelo código do mini curso.



Fonte: Autoria própria.

Eventos

Diretivas permitem exibir e estruturar dados. Mas para tornar a aplicação verdadeiramente interativa, respondendo a cliques, digitação e teclas, usamos **eventos**.

Para escutá-los, usamos a diretiva `v-on` ou sua forma abreviada `@`. A lógica é: "quando este evento acontecer, execute esta ação".

Evento de clique

```
<template>
<button @click="contador++">Clique aqui</button>
<p>Total de cliques: {{ contador }}</p>
</template>

<script>
export default {
        data() {
                return { contador: 0 }
        }
}
</script>
```

Dica

`@click` é a abreviação de `v-on:click`. Assim como `:` abrevia `v-bind:`, o `@` abrevia `v-on:`. Na prática, você sempre verá a forma abreviada.

Chamando métodos

Colocar lógica diretamente no atributo do HTML (como `contador++`) funciona para casos simples, mas logo fica confuso. Para lógicas mais complexas, declare **métodos** na seção `methods` do `<script>` e chame-os no evento:

```
<template>
<button @click="incrementar">Clique</button>
<p>{{ contador }}</p>
</template>

<script>
export default {
        data() {
                return { contador: 0 }
```

```
                },
                methods: {
                        incrementar() {
                                this.contador++;  // "this" acessa as
↪  variáveis do componente
                        }
                }
        }
        </script>
```

Atenção

Dentro de `methods`, sempre use `this.nomeDaVariavel` para acessar as variáveis declaradas em `data()`. Esquecer o `this` é um dos erros mais comuns de quem está começando e vai gerar um erro no console.

Dica

Manter a lógica nos métodos deixa o código mais organizado, fácil de testar e permite reutilizar a função em vários lugares sem repetição.

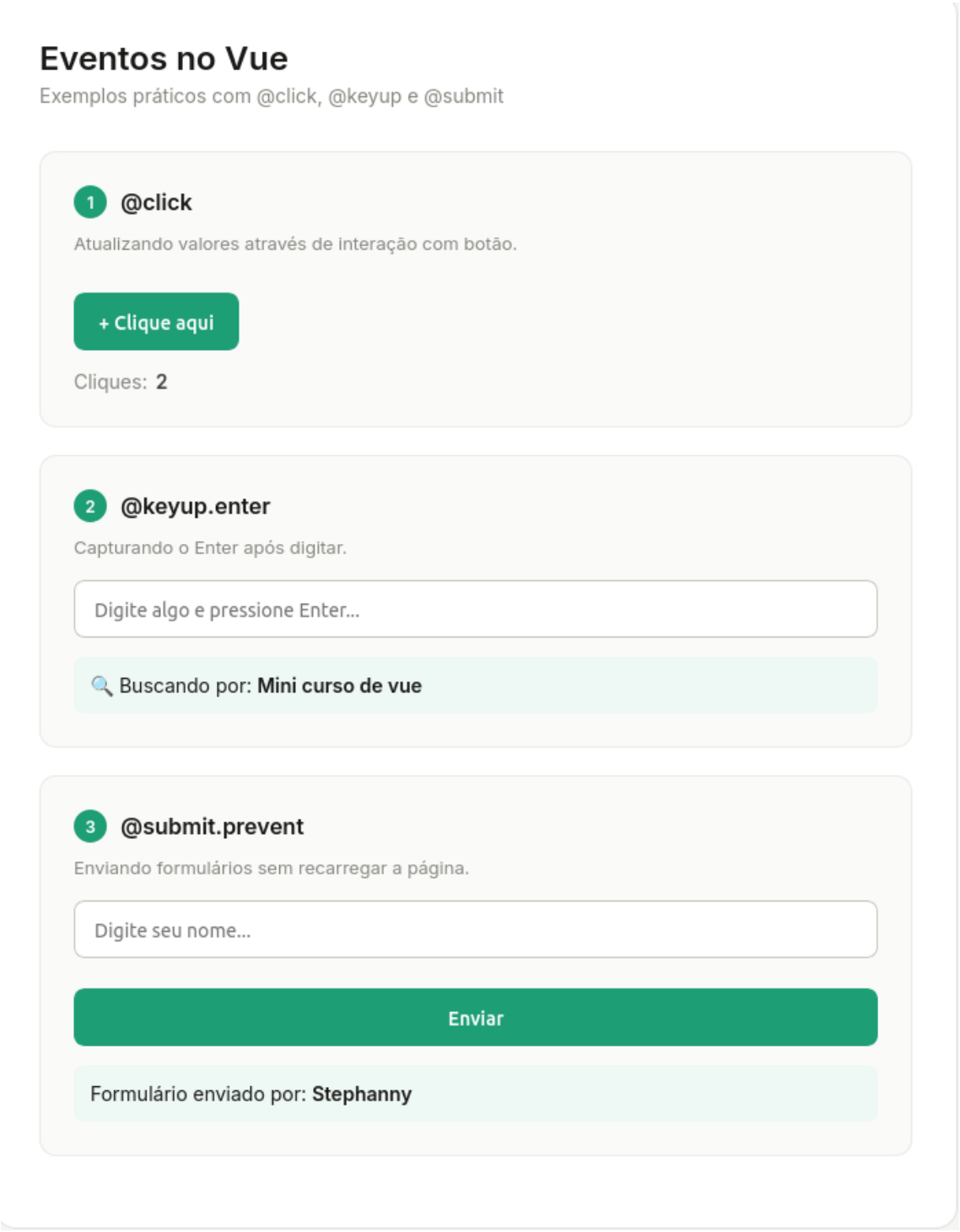
Outros eventos e modificadores

O Vue oferece atalhos para os eventos mais comuns do teclado e formulários:

- `@input` → disparado a cada caractere digitado em um campo.
- `@keyup.enter` → disparado quando o usuário pressiona a tecla Enter.
- `@submit.prevent` → envia o formulário sem recarregar a página (previne o comportamento padrão do HTML).
- `@mouseover` → disparado quando o mouse passa sobre o elemento.

```
        <!-- Busca ao pressionar Enter -->
        <input @keyup.enter="buscar" v-model="consulta">

        <!-- Formulário sem reload da página -->
        <form @submit.prevent="enviar">
        <input v-model="nome">
        <button type="submit">Enviar</button>
        </form>
```

Figura 4.18: Exemplo de eventos exibida pelo código do mini curso.



Fonte: Autoria própria.

Figura 4.19: Fluxo de eventos no Vue.

Fonte: Autoria própria.

Erros comuns e como evitá-los

Antes de partir para o projeto prático, vale revisar os erros mais frequentes entre quem está começando com Vue. Reconhecê-los agora vai poupar muito tempo de depuração.

Atenção

Esquecer de executar `npm install`

Sempre execute `npm install` após clonar ou criar um projeto novo. Sem isso, a pasta `node_modules/` não existirá e o projeto não roda. O sintoma mais comum é o erro `Cannot find module`.

- **Erros de digitação em diretivas:** `v-moodel`, `v-fore` não são reconhecidos pelo Vue. O elemento simplesmente não terá o comportamento esperado, sem aviso de erro. Verifique a ortografia com atenção.
- **Não salvar o arquivo:** o Vite usa HMR (Hot Module Reload), mas o arquivo precisa ser salvo para as mudanças serem detectadas. Use `Ctrl+S` (ou deixe em salvamento automático no VS Code) após cada alteração.
- **Usar `v-for` sem `:key`:** sempre forneça uma chave única para cada item. O Vue alertará no console, mas o código ainda roda, só que de forma potencialmente bugada.
- **Acessar dados não declarados:** certifique-se de que todas as variáveis usadas no template estejam declaradas dentro de `data()`. Variáveis inexistentes resultam em `undefined` ou erros silenciosos.

- **Confundir `v-if` com `v-show`:** use `v-if` para elementos que raramente mudam de visibilidade e `v-show` para alternâncias frequentes (veja a caixa de conceito anterior).
- **Esquecer `this` nos métodos:** dentro de `methods`, acesse variáveis sempre com `this.variavel`, não diretamente pelo nome.

Dica

Abra o console do navegador! Pressione `F12` (ou `Ctrl+Shift+I`) e vá na aba `Console`. O Vue exibe mensagens de aviso e erro muito descritivas, é o seu melhor aliado para depurar problemas.

4.4 Projeto Prático: Lista de Tarefas

Você já conhece os fundamentos do Vue. Agora é hora de aplicar tudo isso em um projeto real. Nesta seção, vamos construir juntos uma **lista de tarefas (To-Do List)** completa, do zero.

Dica

Onde estamos? Esta seção é a prática do curso. Todo conceito que aparece aqui foi explicado na Seção 3. Se encontrar algo que não reconhece, volte à seção correspondente antes de continuar.

Acesso ao projeto

O código base do projeto já está preparado e disponível em um repositório GitHub. Em vez de criar o projeto do zero, você vai **clonar** esse repositório, que contém toda a estrutura necessária para acompanhar a parte prática do curso.

Conceito importante

O que é clonar um repositório?
Clonar significa baixar uma cópia completa do projeto a partir do GitHub para o seu computador. É como fazer o download de todos os arquivos de uma vez, já organizados e prontos para uso. O comando `git clone` faz isso automaticamente.

Passo 1 — Remova o projeto anterior

Se você já criou um projeto Vue durante os exercícios das seções anteriores (por exemplo, a pasta `meu-projeto` ou `projetos-vue`), **apague-o antes de continuar**.

Isso evita conflitos de nomes de pasta e garante que você esteja trabalhando com o repositório correto.

No Windows (Explorador de Arquivos), localize a pasta e pressione `Delete`. Pelo terminal, você pode usar:

```
# No Windows (PowerShell):
Remove-Item -Recurse -Force nome-da-pasta

# No Mac/Linux:
rm -rf nome-da-pasta
```

Atenção

Certifique-se de apagar a pasta certa. Esse comando remove os arquivos permanentemente, sem enviar para a lixeira.

Passo 2 — Clone o repositório do curso

Abra o terminal, navegue até a pasta onde deseja salvar o projeto e execute o comando abaixo:

```
git clone https://github.com/stephannykauane/minicurso-vue.git
```

Isso criará automaticamente uma pasta chamada `minicurso-vue` com todos os arquivos do projeto.

Dica

Caso não tenha o Git instalado, acesse Git [2026], baixe e instale para o seu sistema operacional. Após instalar, feche e reabra o terminal antes de executar o comando acima.

Passo 3 — Acesse a pasta e instale as dependências

Entre na pasta do projeto clonado e instale as dependências necessárias:

```
cd minicurso-vue
npm install
```

Dica

Lembre-se: o repositório clonado **não inclui** a pasta `node_modules/` (ela nunca é enviada ao GitHub). Por isso, o `npm install` é obrigatório toda vez

que você clona um projeto novo. Sem ele, o projeto não roda.

Passo 4 — Inicie o servidor de desenvolvimento

Com as dependências instaladas, inicie o projeto:

```
npm run dev
```

Acesse `http://localhost:5173/` no navegador. Se a página carregar corretamente, o ambiente está pronto!

Atenção

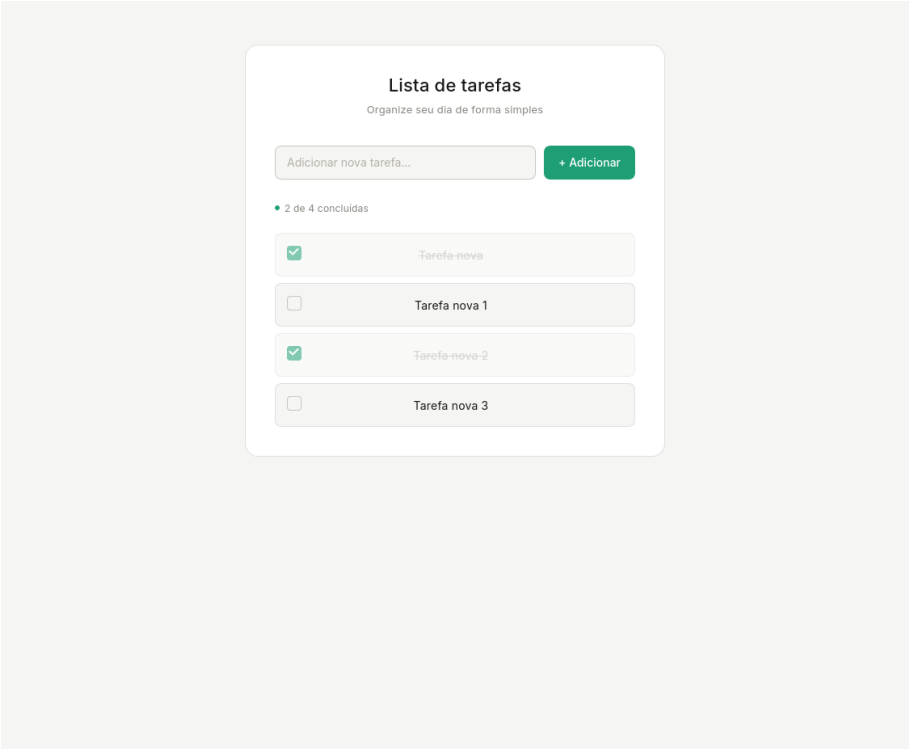
O que já vem no repositório? O projeto clonado contém toda a estrutura da aplicação de lista de tarefas já implementada, incluindo o código completo do `App.vue` que veremos a seguir. A única parte que desenvolveremos juntos ao vivo é a **componentização** (criação do componente filho `BotaoAcao.vue` e sua integração com o componente pai), que será feita na seção 4.4.

Demonstração da aplicação

A aplicação permite:

- Digitar uma nova tarefa em um campo de texto.
- Adicionar a tarefa à lista ao clicar em um botão ou pressionar Enter.
- Marcar tarefas como concluídas com um checkbox.
- Remover tarefas individualmente.
- Ver uma mensagem especial quando a lista estiver vazia.

Figura 4.20: Resultado To-do List.



Fonte: Autoria própria.

Parece simples, mas ela aplica **todos** os conceitos vistos até aqui: reatividade, `v-model`, `v-for`, `v-if` e eventos. Cada funcionalidade existe por um motivo e você já conhece cada peça.

Código completo

```
<template>
<div class="container">
<h2>Lista de Tarefas</h2>

<!-- Campo de entrada + botão -->
<div class="entrada">
<input
v-model="novoItem"
placeholder="Digite uma tarefa..."
@keyup.enter="adicionarItem"
/>
<button @click="adicionarItem">Adicionar</button>
</div>

<!-- Mensagem de lista vazia -->
<p v-if="itens.length === 0" class="vazia">
```

```
17          Nenhuma tarefa adicionada ainda.
18          </p>
19
20          <!-- Lista de tarefas -->
21          <ul v-else>
22          <li
23          v-for="(item, index) in itens"
24          :key="index"
25          :class="{ concluida: item.feita }"
26          >
27          <input type="checkbox" v-model="item.feita" />
28          <span>{{ item.texto }}</span>
29          <button @click="removerItem(index)">Remover</button>
30          </li>
31          </ul>
32          </div>
33          </template>
34
35          <script>
36          export default {
37                  data() {
38                          return {
39                                  novoItem: "",    // texto digitado pelo
    ↪ usuário
40                                  itens: []        // lista de tarefas
    ↪ (começa vazia)
41                          }
42                  },
43                  methods: {
44                          adicionarItem() {
45                                  // Só adiciona se o campo não estiver
    ↪ vazio
46                                  if (this.novoItem.trim() !== "") {
47                                          // Cada item é um objeto com
    ↪ texto e status
48                                          this.itens.push({
49                                                  texto:
    ↪ this.novoItem.trim(),
50                                                  feita: false
51                                          });
52                                          this.novoItem = ""; // limpa o
    ↪ campo após adicionar
```

```
53                                  }
54                          },
55                          removerItem(index) {
56                                  // Remove o item na posição "index" do
    ↪ array
57                                  this.itens.splice(index, 1);
58                          }
59                  }
60          }
61          </script>
62
63          <style scoped>
64          .container {
65                  max-width: 500px;
66                  margin: 40px auto;
67                  font-family: Arial, sans-serif;
68          }
69
70          .entrada {
71                  display: flex;
72                  gap: 8px;
73                  margin-bottom: 16px;
74          }
75
76          input[type="text"], input:not([type]) {
77                  padding: 8px;
78                  flex: 1;
79                  border: 1px solid #ccc;
80                  border-radius: 4px;
81          }
82
83          button {
84                  padding: 8px 14px;
85                  background-color: #42b883;
86                  color: white;
87                  border: none;
88                  border-radius: 4px;
89                  cursor: pointer;
90          }
91
92          button:hover {
93                  background-color: #33a06f;
```

```
94          }
95
96          ul {
97                  margin-top: 16px;
98                  padding-left: 0;
99                  list-style: none;
100         }
101
102         li {
103                 display: flex;
104                 align-items: center;
105                 gap: 10px;
106                 padding: 8px;
107                 border-bottom: 1px solid #ddd;
108         }
109
110         li.concluida span {
111                 text-decoration: line-through;
112                 color: #999;
113         }
114
115         .vazia {
116                 text-align: center;
117                 color: #888;
118         }
119         </style>
```

Explicação do código por partes

Agora que o código está escrito e funcionando no navegador, vamos entender cada parte importante.

1. Campo de entrada e v-model

```
        <input v-model="novoItem" @keyup.enter="adicionarItem">
```

O `v-model` conecta o campo à variável `novoItem`. Tudo que o usuário digitar fica automaticamente salvo nessa variável. O `@keyup.enter` faz com que pressionar Enter também chame `adicionarItem` — o mesmo que clicar no botão.

2. Adicionando tarefas como objetos

```
        this.itens.push({ texto: this.novoItem.trim(), feita: false });
```

Cada tarefa é salva como um **objeto** com dois campos: `texto` (o que foi digitado) e `feita` (se está concluída, começa como `false`). Usar objetos em vez de strings simples permite rastrear o estado de cada item individualmente, o que viabiliza o checkbox da próxima parte.

3. Renderizando a lista com checkbox

```
        <li :class="{ concluida: item.feita }">
        <input type="checkbox" v-model="item.feita" />
        <span>{{ item.texto }}</span>
        </li>
```

O `v-model` no checkbox conecta-o diretamente à propriedade `item.feita`. Quando marcado, essa propriedade vira `true`. O `:class="{ concluida: item.feita }"` aplica a classe CSS `concluida` (que risca o texto) automaticamente sempre que `item.feita` for verdadeiro.

4. Removendo tarefas

```
        this.itens.splice(index, 1);
```

O método `splice(posicao, quantidade)` remove elementos de um array. Aqui, removemos 1 elemento na posição `index`. O Vue detecta a mudança no array e atualiza a lista na tela automaticamente.

Conceito importante

Fluxo mental para pensar em Vue:

1. O usuário interage (digita, clica, marca).
2. Um **dado** é alterado (array, objeto, variável).
3. O Vue atualiza a **interface** automaticamente.

Você nunca precisa escrever "*atualize esse parágrafo*" ou "*adicione esse elemento ao DOM*". Você apenas muda os dados.

Figura 4.21: Fluxo To-do List.

Fonte: Autoria própria.

Momento prático: experimente você mesmo

Antes de continuar, pause e tente fazer as seguintes alterações no projeto. Cada uma exercita um conceito diferente:

- **Cor do botão:** altere `background-color: #42b883` para outra cor de sua escolha. Observe a mudança em tempo real.
- **Texto do placeholder:** mude de "*Digite uma tarefa...*" para algo personalizado.
- **Mensagem de lista vazia:** personalize o texto exibido quando não há tarefas.

Dica

O servidor `npm run dev` atualiza o navegador automaticamente a cada vez que você salva o arquivo (`Ctrl+S`). Você verá as mudanças em tempo real, isso é o **Hot Module Reload** do Vite em ação!

Componentização: organizando o projeto

Agora que a aplicação funciona em um único arquivo, vamos dar um passo além: separar partes reutilizáveis em **componentes próprios**. Essa é uma das práticas

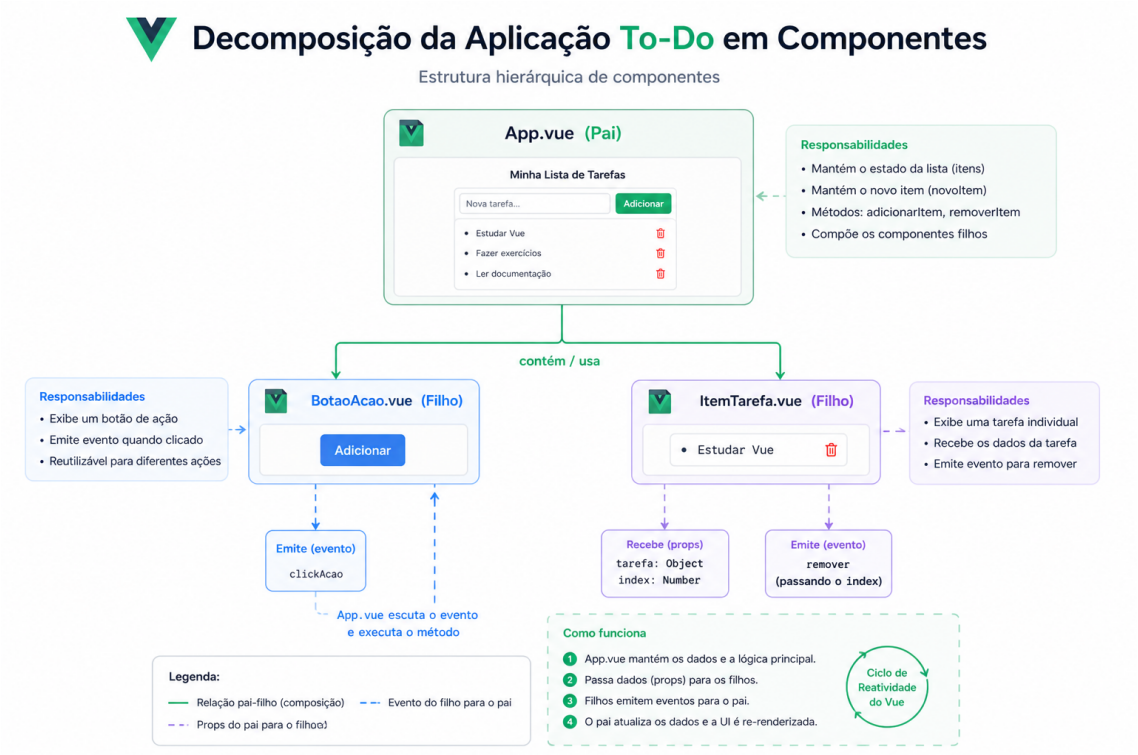
mais importantes do desenvolvimento Vue em projetos reais.

A ideia é simples: em vez de ter um único arquivo gigante, você divide a interface em peças menores e independentes. Cada peça é um componente com sua própria lógica e estilo.

Atenção

Atividade ao vivo! Esta é a parte que implementaremos juntos durante o mini curso. Diferentemente do restante do projeto (que já veio pronto no repositório), o componente `BotaoAcao.vue` e sua integração com o `App.vue` serão criados por você agora, passo a passo.

Figura 4.22: Decomposição da aplicação em componentes.



Fonte: Autoria própria.

Criando um componente filho

Vamos criar um botão reutilizável. Siga os passos:

1. Na pasta `src/components/`, crie um novo arquivo chamado `BotaoAcao.vue`.
2. Cole o código abaixo:

```
1       <template>
2       <!-- Quando clicado, emite o evento "acao" para o componente pai
        ↪  -->
```

```
        <button class="btn" @click="$emit('acao')">{{ texto }}</button>
        </template>

        <script>
        export default {
                props: {
                        // "texto" é uma propriedade recebida do componente pai
                        texto: {
                                type: String,
                                default: "Clique"
                        }
                }
        }
        </script>

        <style scoped>
        .btn {
                padding: 8px 14px;
                background-color: #42b883;
                color: white;
                border: none;
                border-radius: 4px;
                cursor: pointer;
        }
        </style>
```

Dica

O que são props? Props (abreviação de *properties*) são como parâmetros de uma função: o componente pai os envia, e o filho os recebe para personalizar seu comportamento ou aparência. Aqui, o pai enviará o texto do botão via `texto`.

Usando o componente no pai

Agora, no `App.vue` (o componente pai), importe e use o `BotaoAcao`:

```
        <template>
        <div>
        <!-- Usa o componente duas vezes com textos diferentes -->
        <BotaoAcao texto="Adicionar"   @acao="adicionarItem" />
```

```
5         <BotaoAcao texto="Limpar tudo" @acao="limparLista"    />
6         </div>
7         </template>
8
9         <script>
10        // 1. Importa o componente pelo caminho do arquivo
11        import BotaoAcao from './components/BotaoAcao.vue'
12
13        export default {
14                // 2. Registra o componente para poder usá-lo no template
15                components: { BotaoAcao },
16                methods: {
17                        adicionarItem() { /* código de adicionar */ },
18                        limparLista()   { /* código de limpar */ }
19                }
20        }
21        </script>
```

Conceito importante

Comunicação entre componentes:

- **Props (`:prop="valor"`)** → o componente **pai envia dados para o filho**. É como configurar o filho antes de exibi-lo.
- **Emit (`$emit('evento')`)** → o componente **filho notifica o pai** sobre ações. O filho não faz nada sozinho, ele apenas avisa o pai, e o pai decide o que fazer.

Pense assim: o pai *configura* o filho via props, e o filho *avisa* o pai via emit quando algo acontece.

Figura 4.23: Fluxo To-do List.

Fonte: Autoria própria.

4.5 Revisão do que foi aprendido

Parabéns por chegar até aqui! Neste mini curso, você aprendeu a:

- Entender o que é o Vue.js e como ele difere do JavaScript puro.
- Entender o papel do Vite no desenvolvimento com Vue.
- Compreender o conceito de **reatividade** e o fluxo usuário → dados → interface.
- Configurar o ambiente com Node.js e VS Code.
- Criar um projeto Vue com Vite (`npm create vite@latest`).
- Estruturar componentes com `<template>`, `<script>` e `<style>`.
- Usar as principais diretivas: `v-model`, `v-if/else`, `v-for`, `v-bind`, `v-show`.
- Capturar eventos com `@click`, `@keyup.enter`, `@submit.prevent`.
- Desenvolver uma lista de tarefas funcional com checkbox.
- Criar componentes reutilizáveis com *props* e *emit*.

Próximos passos: melhorias no projeto

Para continuar evoluindo a aplicação desenvolvida no curso, sugerimos:

- **Persistência de dados:** salvar as tarefas no `localStorage` para que não se percam ao recarregar a página.
- **Filtros:** adicionar botões para exibir "Todas", "Pendentes" e "Concluídas".
- **Contador:** mostrar quantas tarefas restam (ex.: "3 de 5 concluídas").
- **Animações:** usar `<Transition>` do Vue para animar a entrada e saída de itens.
- **Vue Router:** adicionar navegação entre páginas (ex.: tela inicial e tela de tarefas).
- **Pinia:** centralizar o estado da aplicação em uma store global.

Dica

A documentação oficial do Vue.js é excelente e está disponível em português: Vue.js [2026]

4.6 Declaração de uso de IA generativa

☐ NÃO utilizei ferramentas de IA generativa para produzir, reescrever ou expandir trechos do texto do meu mini-curso. Eventuais correções automáticas de ortografia/gramática do editor de texto não se enquadram nesta declaração.

☑ UTILIZEI ferramentas de IA generativa de forma ética e transparente, exclusivamente como apoio redacional (ex.: revisão gramatical, melhoria de clareza, sugestão de organização), sem terceirização do conteúdo intelectual. Declaro que: (a) informei o uso; (b) revisei integralmente e sou responsável(a) pelo texto final; (c) conferi a veracidade de dados, citações e referências, garantindo que não há referências inventadas, trechos sem fonte ou conteúdo não verificável.

Se marcou acima a opção UTILIZEI, descreva de forma objetiva:

- **Ferramenta(s)/versão(ões):** Chat gpt GPT-5.5;
- **Seções/trechos impactados:** Da Seção 1 à Seção 4;
- **Finalidade(s):** Na geração de imagens para compor diagramas e mapas mentais e códigos de exemplos.

4.7 Bibliografia

Git (2026). Git - install. https://git-scm.com/downloads. Acessado em 16 maio 2026.

Node.js (2026). Node.js - baixar node.js. https://nodejs.org/pt-br/download. Acessado em 16 maio 2026.

Visual Studio Code (2026). Visual studio code. https://code.visualstudio.com/. Acessado em 16 maio 2026.

Vue.js (2026). Vue.js - the progressive javascript framework. https://vuejs.org/. Acessado em 16 maio 2026.

Minicurso 5

Utilização do Shell e Bash para organização e automação de tarefas

Marcos Santos da Silva e Helder Oliveira Gomes De Souza

Resumo

Este minicurso tem por objetivo capacitar participantes iniciantes a navegar, manipular arquivos e escrever scripts simples em Bash, melhorando produtividade e autonomia em ambientes Linux.

5.1 Introdução

O sistema operacional Linux é amplamente adotado em servidores, infraestrutura de redes, desenvolvimento de software e ambientes acadêmicos, destacando-se por sua estabilidade, segurança e flexibilidade. Baseado em software livre e de código aberto, o Linux permite que usuários e organizações tenham maior controle sobre o sistema, além de possibilitar customizações conforme suas necessidades específicas.

Nesse contexto, o domínio do Shell e da linguagem Bash torna-se essencial para profissionais da área de tecnologia da informação. A utilização do terminal permite uma interação direta com o sistema operacional, proporcionando maior eficiência na execução de tarefas, automação de processos repetitivos e gerenciamento de recursos de forma otimizada.

Além disso, o conhecimento em Linux é altamente valorizado no mercado de trabalho, sendo uma competência fundamental para áreas como administração de sistemas, segurança da informação, computação em nuvem e desenvolvimento. Sua ampla utilização em servidores e sistemas embarcados reforça a importância de compreender seus conceitos básicos e avançados.

Dessa forma, este minicurso tem como objetivo introduzir os principais conceitos do sistema operacional Linux, abordando desde operações básicas no terminal

até a criação de scripts em Bash, proporcionando aos participantes uma base sólida para utilização prática e aprofundamento futuro na área.

5.2 Comparação entre Sistemas Operacionais

Os sistemas operacionais mais utilizados atualmente incluem Linux, Windows e macOS. Cada um possui características próprias relacionadas à usabilidade, desempenho, segurança e propósito de uso.

O Windows é amplamente utilizado em ambientes domésticos e corporativos, sendo conhecido por sua interface gráfica intuitiva e grande compatibilidade com softwares comerciais. No entanto, é um sistema proprietário e possui menor flexibilidade para personalização.

O macOS, desenvolvido pela Apple, é utilizado exclusivamente em computadores da marca. Destaca-se pela integração entre hardware e software, estabilidade e interface refinada. Assim como o Windows, é um sistema proprietário.

O Linux, por sua vez, é um sistema de código aberto, permitindo livre utilização, modificação e distribuição. É amplamente utilizado em servidores, desenvolvimento de software, computação científica e ambientes educacionais.

Principais vantagens do Linux:

- Código aberto e gratuito;
- Alta segurança e menor incidência de vírus;
- Grande estabilidade e desempenho;
- Flexibilidade e personalização;
- Forte uso em servidores e infraestrutura de redes.

Devido a essas características, o Linux é uma excelente escolha para aprendizado técnico e automação de tarefas.

5.3 Público-alvo

Estudantes de graduação em Ciência da Computação, Sistemas de Informação ou áreas afins, bem como profissionais interessados em adquirir conhecimentos básicos de administração e automação em Linux.

5.4 Pré-requisitos

- Noções básicas de uso de computador;
- Acesso a um computador com Linux instalado (VM ou WSL recomendado);
- Editor de texto simples e terminal.

5.5 Carga Horária

Carga horária total: 90 minutos.

5.6 Roteiro

1. Abertura e objetivos — 10 min
2. Introdução ao Linux e Shell — 10 min
3. Estrutura de diretórios — 20 min
4. Manipulação de arquivos — 10 min
5. Scripts Bash — 20 min
6. Redirecionamento e pipes — 10 min
7. Atividade prática — 10 min

5.7 Metodologia

A abordagem combina exposição teórica com práticas orientadas.

5.8 Fundamentação Teórica

Esta seção apresenta os conceitos fundamentais necessários para a compreensão do uso do sistema operacional Linux, com foco na interação por meio do terminal e na automação de tarefas utilizando scripts Bash.

Sistema Operacional Linux

O Linux é um sistema operacional baseado em software livre e de código aberto, amplamente utilizado em servidores, ambientes acadêmicos e sistemas embarcados. Sua arquitetura é baseada no modelo Unix, priorizando estabilidade, segurança e flexibilidade.

Diferente de sistemas proprietários, o Linux permite que usuários tenham acesso ao código-fonte, possibilitando modificações e adaptações conforme a necessidade.

Kernel e Shell

O sistema Linux é composto por diferentes camadas. O **Kernel** é o núcleo do sistema operacional, responsável por gerenciar recursos como memória, processos e dispositivos de hardware.

Já o **Shell** é a interface que permite a interação do usuário com o sistema. Ele interpreta comandos digitados e solicita ao Kernel a execução das ações correspondentes.

Entre os shells disponíveis, o Bash (Bourne Again Shell) é o mais utilizado nas distribuições Linux.

Interface de Linha de Comando (CLI)

A Interface de Linha de Comando (CLI) é um meio de interação com o sistema por meio de comandos textuais. Diferente das interfaces gráficas (GUI), a CLI oferece maior controle, precisão e eficiência na execução de tarefas.

O uso da CLI é essencial em ambientes profissionais, especialmente em administração de sistemas e automação.

Sistema de Arquivos

O sistema de arquivos do Linux é organizado de forma hierárquica a partir do diretório raiz ("/"). Todos os arquivos e diretórios estão contidos dentro dessa estrutura.

Essa organização permite padronização e facilita a administração do sistema. Diretórios como `/home`, `/etc` e `/var` possuem funções específicas dentro do sistema.

Permissões de Arquivos

No Linux, cada arquivo possui permissões que definem quem pode acessá-lo e de que forma. Essas permissões são divididas em leitura (r), escrita (w) e execução (x), aplicadas ao usuário, grupo e outros.

O controle de permissões é fundamental para a segurança do sistema e para o gerenciamento adequado dos recursos.

Automação com Scripts Bash

Scripts Bash são arquivos que contêm sequências de comandos que podem ser executados automaticamente. Eles são amplamente utilizados para automatizar tarefas repetitivas, como organização de arquivos, backups e execução de rotinas administrativas.

A utilização de scripts permite ganho de produtividade e redução de erros operacionais.

Redirecionamento e Pipes

O Linux permite a combinação de comandos por meio de redirecionamento e pipes. O redirecionamento possibilita enviar a saída de um comando para arquivos ou outros destinos.

Já os pipes permitem que a saída de um comando seja utilizada como entrada para outro, possibilitando a criação de fluxos de processamento de dados.

Agendamento de Tarefas com CRON

O CRON é um serviço responsável por executar comandos automaticamente em horários definidos. Ele é amplamente utilizado para automação de tarefas em servidores e sistemas de produção.

O agendamento é configurado por meio do comando `crontab`, permitindo definir rotinas periódicas de execução.

Importância da Automação

A automação de tarefas é uma prática essencial na área de tecnologia da informação. Ela permite reduzir o esforço manual, minimizar erros e aumentar a eficiência operacional.

No contexto do Linux, a automação é facilitada pelo uso do Shell e de scripts Bash, tornando-se uma habilidade fundamental para profissionais da área.

Objetivos

Capacitar os participantes a utilizarem o sistema operacional Linux de forma prática, e fácil compreendendo seus conceitos básicos, navegação no terminal e execução de tarefas comuns do dia a dia.

Introdução ao Linux e Shell

O que é um Shell? O Shell é um programa que interpreta comandos — ou seja, ele é a interface entre você e o sistema operacional Linux Quando você digita um comando, o Shell entende o que foi escrito e executa a ação no sistema Um exemplo muito comum é o Bash, que é um dos shells padrão em várias distribuições Linux Em resumo, O Shell é o "intérprete de comandos". O que será um Terminal? O Terminal é o programa (janela/aplicativo) que permite que você interaja com o Shell Ele é apenas a interface visual onde você digita os comandos Exemplos de terminais incluem: Terminal do Ubuntu GNOME Terminal Konsole O Terminal é a "porta de entrada" para o Shell.

Estrutura de diretórios

Imagine uma árvore invertida. A raiz está no topo, e todos os outros arquivos e pastas são galhos que saem dela.

Os Diretórios Principais:

/bin (Binaries): Contém os arquivos executáveis essenciais do sistema que todos os usuários podem usar (como os comandos ls, cp e cd).

/boot: Onde ficam os arquivos necessários para inicializar o computador, incluindo o Kernel do Linux.

/dev (Devices): No Linux, tudo é um arquivo. Este diretório contém arquivos que representam o hardware (seu HD, mouse, portas USB, etc.).

/etc: O "painel de controle"do sistema. Aqui ficam quase todos os arquivos de configuração do computador e dos serviços.

/home: É onde moram os usuários comuns. Se o seu usuário é "joao", sua pasta será /home/joao. É o único lugar (além do /tmp) onde você tem permissão total por padrão.

/root: O diretório pessoal do superusuário (o administrador). Não confunda com a raiz /.

/sbin (System Binaries): Semelhante ao /bin, mas contém comandos destinados apenas ao administrador para manutenção do sistema.

/tmp (Temporary): Arquivos temporários criados por programas. Geralmente são apagados quando você reinicia.

/usr (User System Resources): A maior parte dos programas instalados fica aqui. É como o "Program Files"do Windows.

/var (Variable): Arquivos que mudam de tamanho constantemente, como logs do sistema, bancos de dados e filas de impressão.

Manipulação de arquivos

Nesta etapa do nosso mini curso, vamos focar em como você interage com os arquivos de fato. No Linux, a manipulação de arquivos é feita majoritariamente através de comandos que seguem uma lógica simples: comando + origem + destino. Criando e Editando Arquivos Antes de manipular, precisamos de conteúdo. No terminal, existem várias formas de ”dar vida”a um arquivo:

Scripts Bash

Um Script Bash é basicamente um arquivo de texto que contém uma sequência de comandos que o Linux executará um após o outro. É a forma mais poderosa de automatizar tarefas repetitivas no sistema.

Pense no script como uma ”receita de bolo”: você define os ingredientes (variáveis) e os passos (comandos), e o computador os segue fielmente.

Redirecionamento e pipes

. o Linux, o conceito de Redirecionamento e Pipes (canalizações) é o que permite combinar comandos simples para realizar tarefas complexas. A filosofia do Unix é: ”Crie programas que façam apenas uma coisa e a façam bem. Crie programas para trabalharem juntos.”

5.9 Atividades Práticas

Atividade 1 — Organização de diretórios

Objetivo: Familiarizar o participante com a estrutura de diretórios e comandos básicos.

```
1 #!/bin/bash
2 mkdir projeto_linux
3 cd projeto_linux
4 mkdir documentos imagens scripts backups
5 touch documentos/relatorio.txt
6 touch imagens/foto1.png
7 touch scripts/script.sh
8 mv documentos/relatorio.txt backups/
```

Atividade 2 — Permissões de arquivos

```
#!/bin/bash
ls -l
chmod 755 scripts/script.sh
chmod 644 documentos/notas.txt
```

Atividade 3 — Script de organização de arquivos

```
#!/bin/bash

if [ -z "$1" ]; then
  echo "Uso: ./organizar.sh .extensao"
  exit 1
fi

EXTENSAO=$1
DATA=$(date +%F)
PASTA="backup_$DATA"

mkdir -p "$PASTA"

for arquivo in *"$EXTENSAO"; do
  if [ -f "$arquivo" ]; then
    mv "$arquivo" "$PASTA/"
  fi
done

echo "Organização concluída!"
```

Atividade 4 — Pipes e redirecionamento

```
#!/bin/bash
ls > lista.txt
cat lista.txt | grep ".txt"
ls | sort
history
```

Atividade 5 — Automação com CRON

Objetivo: Automatizar execução de tarefas no Linux.

```
#!/bin/bash
crontab -e

# Executa script todos os dias às 02:00
0 2 * * * /home/usuario/scripts/organizar.sh .txt
```

Explicação:

- 0 2 * * * → 02:00 da manhã todos os dias
- Caminho deve ser absoluto

Atividade 6 — Script de Backup

Objetivo: Criar cópia de segurança de diretórios.

```
#!/bin/bash

ORIGEM="$HOME/projeto_linux"
DESTINO="$HOME/backups"
DATA=$(date +%F)

mkdir -p "$DESTINO"

cp -r "$ORIGEM" "$DESTINO/backup_$DATA"

echo "Backup realizado com sucesso em $DESTINO/backup_$DATA"
```

Atividade 7 — Backup de HD ou diretório com compactação

```
#!/bin/bash

ORIGEM="/home/usuario/projeto_linux"
DESTINO="/home/usuario/backups"
DATA=$(date +%F)

```

```
mkdir -p "$DESTINO"

tar -czvf "$DESTINO/backup_$DATA.tar.gz" "$ORIGEM"

echo "Backup compactado criado!"
```

Atividade 8 — Apelidos (Aliases)

Objetivo: Criar comandos personalizados.

```
#!/bin/bash
alias ll='ls -la'
alias atualizar='sudo apt update && sudo apt upgrade -y'
alias limpar='clear'
alias voltar='cd ..'
```

Salvar permanentemente:

```
#!/bin/bash
nano ~/.bashrc
# adicionar aliases no final

source ~/.bashrc
```

5.10 Resultados Esperados

- Compreender a estrutura básica do sistema Linux e sua organização de diretórios;
- Navegar e manipular arquivos e diretórios utilizando comandos do Shell;
- Gerenciar permissões de arquivos e diretórios;
- Criar e executar scripts simples em Bash;
- Automatizar tarefas rotineiras utilizando scripts;
- Utilizar redirecionamento e pipes para processamento de dados;
- Implementar rotinas automatizadas com o uso do CRON;

- Criar rotinas de backup de arquivos e diretórios;
- Utilizar aliases para otimizar o uso do terminal e aumentar a produtividade.

5.11 Avaliação

Avaliação formativa baseada na prática e entrega do script.

5.12 Declaração de uso de IA generativa

☑ NÃO utilizei ferramentas de IA generativa para produzir, reescrever ou expandir trechos do texto do meu mini-curso. Eventuais correções automáticas de ortografia/gramática do editor de texto não se enquadram nesta declaração.

☐ UTILIZEI ferramentas de IA generativa de forma ética e transparente, exclusivamente como apoio redacional (ex.: revisão gramatical, melhoria de clareza, sugestão de organização), sem terceirização do conteúdo intelectual. Declaro que: (a) informei o uso; (b) revisei integralmente e sou responsável(a) pelo texto final; (c) conferi a veracidade de dados, citações e referências, garantindo que não há referências inventadas, trechos sem fonte ou conteúdo não verificável.

Se marcou acima a opção UTILIZEI, descreva de forma objetiva:

- **Ferramenta(s)/versão(ões):** NÃO SE APLICA;
- **Seções/trechos impactados:** NÃO SE APLICA;
- **Finalidade(s):** NÃO SE APLICA.

5.13 Bibliografia

Free Software Foundation (2026a). Bash reference manual. https://www.gnu.org/software/bash/manual/. Acesso em: 01 junho 2026.

Free Software Foundation (2026b). Documentação do comando `cp`. https://www.gnu.org/software/coreutils/. Acesso em: 01 junho 2026.

Free Software Foundation (2026c). Documentação do comando `tar`. https://www.gnu.org/software/tar/. Acesso em: 01 junho 2026.

Kerrisk, M. (2026). Cron documentation (crontab). https://man7.org/linux/man-pages/man5/crontab.5.html. Acesso em: 01 junho 2026.

Nemeth, E., Snyder, G., and Hein, T. (2007). *Manual Completo do Linux*. Pearson.

Nemeth, E., Snyder, G., and Hein, T. (2010). *Unix and Linux System Administration Handbook*. Prentice Hall.

Shotts, W. E. (2026). *The Linux Command Line: A Complete Introduction*. No Starch Press.

TLDP (2026). The linux documentation project. https://tldp.org/. Acesso em: 01 junho 2026.

Minicurso 6

Detecção de objetos com YoloV8

Camila Gabriele Figueredo dos Santos e Kaike Leles Lamonier Soares

Resumo

Este minicurso apresenta, de forma prática e acessível, como funciona a detecção de objetos utilizando o YOLOv8, aplicando os conceitos em casos reais, como a identificação de personagens em cenas de jogos e vagas de estacionamento. A atividade mostrará o passo a passo de um projeto de visão computacional, abordando desde a organização do dataset e marcação nas imagens, até o treinamento, destacando diferenças de performance em hardware (GPU vs. CPU) e configuração de ambiente.

6.1 Introdução

O objetivo principal deste minicurso é apresentar, de forma prática e acessível, como funciona a detecção de objetos utilizando a arquitetura YOLOv8. Durante o minicurso, os conceitos serão aplicados em casos práticos do mundo real, utilizando como exemplo a identificação do personagem Waldo e o monitoramento de vagas de carros.

Os participantes acompanharão o fluxo de trabalho completo de um projeto de visão computacional, que abrange desde a organização e preparação do dataset, configuração do ambiente de desenvolvimento, até as etapas de treinamento, inferência e análise das métricas do modelo treinado.

6.2 Público-Alvo e Pré-Requisitos

Público-Alvo

O conteúdo foi desenhado para atingir o seguinte perfil:

- Alunos interessados na área de Inteligência Artificial.

- Pessoas com conhecimento básico em informática que desejam compreender aplicações práticas de visão computacional.

Pré-Requisitos dos Participantes

Para o acompanhamento adequado das atividades, espera-se que os participantes possuam:

- Noções básicas em informática e uso de computadores;
- Noção básica de programação na linguagem Python;
- Interesse em Inteligência Artificial e detecção de objetos.

6.3 Estrutura do Minicurso e Cronograma

A atividade foi estruturada para ser concluída em um tempo máximo de 90 minutos, sendo dividida em uma fundamentação teórica objetiva seguida por uma extensa aplicação prática.

Conteúdo Teórico (35 minutos)

A base teórica tem o propósito de nivelar os conhecimentos necessários para a prática, seguindo o roteiro abaixo:

- O que é Visão Computacional (5 min)
- Diferença conceitual entre classificação e detecção (6 min)
- Conceito e estruturação da arquitetura YOLO (7 min)
- Organização de *dataset*: divisão em conjuntos de *train*, *val* e *test* (4 min)
- Conceito de *bounding box* (3 min)
- Apresentação da plataforma Roboflow (2 min)
- Métricas principais para avaliação: *Precision*, *Recall* e *mAP* (8 min)

Conteúdo Prático (55 minutos)

A segunda etapa focará na execução (*hands-on*) do fluxo de detecção:

- Apresentação do problema a ser resolvido utilizando o YOLO para detecção (5 min);

- Marcação das *bounding boxes* com base em um *dataset* pré-selecionado (10 min);
  - *Datasets* predefinidos: Roboflow [2024b], Delgado [2023], Roboflow [2024a];
- Exportação dos datasets classificados e formatados (1 min);
- Configuração do ambiente de desenvolvimento local (5 min);
- Execução do código de divisão e treinamento do *dataset* (15 min);
- Análise das métricas geradas no pós-treinamento (10 min);
- Teste e inferência do modelo utilizando imagens reais (9 min).

6.4 Configuração de Ambiente e Hardware

Ambiente Virtual e Versões

Para garantir que as dependências do projeto não conflitem com o sistema operacional, utilizamos o conceito de *Virtual Environment* (Venv). O ambiente virtual isola as bibliotecas Python necessárias e assim não há interferências nas bibliotecas globais. Funciona como uma ”bolha”segura que impede que pacotes de um projeto afetem outros. As versões exatas testadas e documentadas estão descritas na Tabela 6.1.

No Código-fonte 72, demostramos as linhas de código necessárias para a criação do .venv (Virtual Environment) no ambiente do Windows, para outros sistemas operacionais e/ou editores de texto consulte a documentação da linguagem em Foundation [2026].

Código-fonte 72 Criação e Ativação do venv no Windows.

```
$ python -m venv .venv #Criação do Ambiente
$ .venv\Scripts\activate #Ativação do Ambiente
```

Fonte: Elaborado pelos autores

Tabela 6.1: Especificações de Software e Versões Recomendadas.

Componente	Versão Utilizada
Python	3.11.9
Ultralytics (YOLO)	8.4.41
*PyTorch	2.5.1 (Com suporte CUDA 2.5.1+cu121)
Ipykernel	7.2.0
CUDA Toolkit	12.1

Fonte: Elaborado pelos autores.

Visando demonstrar o impacto do hardware utilizado para realização do treinamento, realizamos testes em três dispositivos com configurações diferentes especificados na Tabela 6.2.

Tabela 6.2: Especificações de Hardware Utilizadas.

Dispositivo	Processador	Memória RAM	GPU (VRAM)
Nitro V15	Intel Core i5-13420H	16GB	RTX 4050 (6GB)
Lab. Faculdade	Intel Core i7-9700T	16GB	Apenas CPU
Google Colab	Standard Xeon	12GB	T4 GPU (16GB)

Fonte: Elaborado pelos autores.

Diferenças de SO: Windows vs. Linux

A configuração do ambiente virtual varia dependendo do sistema operacional:

- **Windows:** A ativação do Venv é feita via PowerShell. Muitas vezes é necessário alterar a *ExecutionPolicy* do Windows para permitir a execução de scripts.
- **Linux (Ubuntu/Debian):** Nativamente mais ágil para gestão de processos, porém exige a instalação prévia do pacote `python3-venv`. Um detalhe crucial no Linux: ao utilizar o VS Code com cadernos interativos (.ipynb), é **obrigatório** instalar a biblioteca `ipykernel` dentro do Venv para que o editor consiga interpretar e executar as células.

Hardware, GPU e Linguagem CUDA

Treinar redes neurais exige cálculo paralelo intensivo.

- **Placas de Vídeo e CUDA:** O treinamento otimizado do YOLOv8 requer a tecnologia **CUDA**, uma linguagem de computação paralela exclusiva das GPUs da **NVIDIA** (como a RTX 4050 utilizada para o treinamento).
- **Sem GPU (Apenas CPU):** O treinamento ocorrerá no processador central. Funciona perfeitamente, porém de forma sequencial, tornando o processo substancialmente mais lento.

.py vs .ipynb

Em ambientes python para treinamento de modelos de inteligência artificial, utilizamos dois formatos de arquivos:

- **.ipynb (Jupyter Notebook):** Usado neste minicurso, é um ambiente "orgânico"que mescla blocos de texto, imagens e células de código executáveis independentemente. Excelente para testes e análise visual.
- **.py (Script Python):** Arquivo de código puro e direto, sem separações. Usado na fase de "produção", quando o modelo já está validado e o software final precisa ser executado de ponta a ponta sem interrupções.

6.5 Arquitetura YOLOv8 e Estrutura de Pastas

Existem diversas arquiteturas de visão computacional (como *Faster R-CNN* ou *SSD*). Utilizamos o **YOLO** (*You Only Look Once*) pois ele processa a imagem inteira em uma única rede neural de estágio, como embasado por Sohan et al. [2024]. Isso garante uma velocidade extraordinária (tempo real) mantendo uma altíssima precisão. Pode obter o detalhamento das características da YOLO através de sua documentação disponível em Ultralytics [2026b] e Ultralytics [2026a]

Ao executar o YOLOv8, ele cria automaticamente uma estrutura de diretórios para salvar o progresso:

- **Pasta runs/detect/train/:** Onde ficam salvos os gráficos gerados e os resultados das validações.
- **Arquivo de Pesos (best.pt):** Localizado dentro da pasta `../weights/`, este arquivo é o "cérebro"da IA. Ele armazena o aprendizado final, guardando em seu conteúdo os melhores resultados alcançados durante o processo de aprendizado. Após o treinamento, usamos apenas este arquivo para fazer detecções, sem precisar retreinar o modelo.

O Arquivo de Configuração (.yaml)

O arquivo `.yaml`, localizado junto dos demais arquivos do dataset, é o mapa de instruções para o treinamento. Ele informa à arquitetura do YOLO exatamente onde estão os diretórios contendo as imagens, a quantidade total de classes (`nc`) e os rótulos exatos (`names`) que o modelo deve identificar.

No Código-fonte 73, apresentamos a estrutura do arquivo gerado para o *dataset* do Waldo que utilizamos durante o minicurso:

Código-fonte 73 Estrutura do arquivo de configuração data.yaml.

```yaml
train: ../train/images
val: ../valid/images
test: ../test/images

nc: 1
names: ['waldorotation']

roboflow:
  workspace: WorkSpaceName
  project: waldo-qa5u7-b7ldd
  version: 2
  license: CC BY 4.0
  url: https://universe.roboflow.com/
```

Fonte: Arquivo *data.yaml* adaptado de Roboflow [2024b].

Estrutura de Resultados Gerados (Pasta runs)

Após a execução de um treinamento, a biblioteca Ultralytics gera automaticamente uma árvore de diretórios (Pastas e subpastas contendo arquivos) organizada para armazenar todos os artefatos, métricas e pesos do modelo. Por padrão, em tarefas de detecção, esses resultados são salvos no caminho `runs/detect/train/`. Caso múltiplos treinamentos sejam executados no mesmo ambiente, o YOLO cria sequencialmente pastas numeradas como `train-2/`, `train-3/`, e assim por diante.

Compreender o conteúdo gerado dentro desse diretório é fundamental para a análise pós-treinamento. Os principais arquivos e pastas encontrados são:

- **`weights/`:** Subpasta crucial que contém os arquivos binários com os ”pesos”da rede neural. Aqui ficam armazenados o `best.pt` (modelo que obteve as melhores métricas gerais durante a fase de validação) e o `last.pt` (modelo salvo na exata última época do treinamento, útil caso o processo seja interrompido e precise ser retomado).

- **`args.yaml`:** Arquivo que registra todos os hiperparâmetros e argumentos utilizados no momento do treinamento (como tamanho da imagem, número de épocas, taxa de aprendizado, configurações de *data augmentation*, etc.). Este arquivo garante a reprodutibilidade exata do experimento.

- **`results.csv` e `results.png`:** Planilha e gráfico que consolidam as métricas de perda (*loss*), Precisão, Recall e mAP ao longo de todas as épocas. São os arquivos mais importantes para visualizar a curva de aprendizado da inteligência artificial.

- **`confusion_matrix.png`:** A matriz de confusão gerada automaticamente com base no conjunto de validação, essencial para identificar falsos positivos e falsos negativos e entender onde o modelo está "se confundindo".
- **`train_batch.jpg`:** Imagens de amostra em formato de mosaico mostrando os lotes (*batches*) de dados exatos que foram alimentados na rede neural. Elas incluem as *bounding boxes* e os rótulos originais para conferência visual de que o *dataset* foi importado e lido corretamente.
- **`val_batch_pred.jpg`:** Imagens do conjunto de validação sobrepostas com as predições feitas pelo modelo em treinamento. Permitem uma comparação visual imediata entre o que o modelo previu e o rótulo real (ground truth).
- **Curvas de Avaliação (`F1_curve.png`, `PR_curve.png`, `P_curve.png`, `R_curve.png`):** Conjunto de gráficos detalhados que relacionam a Precisão e a Revocação com o Nível de Confiança (*Confidence Score*) do modelo, vitais para o ajuste fino na etapa de inferência.

6.6 Desenvolvimento Prático: Detecção de Objetos

A etapa prática materializa os conceitos discutidos. Utilizaremos uma metodologia que permite que os conhecimentos adquiridos sejam replicados em qualquer área.

Preparação e Curadoria de Dados (Datasets)

Para o desenvolvimento e treinamento dos nossos modelos de detecção de objetos com o YOLOv8 durante este minicurso, utilizaremos três conjuntos de dados (*datasets*) públicos, hospedados na plataforma Roboflow Universe. A escolha destes *datasets* foi feita para garantir uma progressão no nível de dificuldade e abordar diferentes cenários práticos de Visão Computacional: desde a identificação de personagens, busca por objetos ocultos em cenários complexos, até aplicações reais de cidades inteligentes.

Todos os *datasets* passaram por uma etapa de pré-processamento padrão de Auto-orientação (*Auto-orient*), que garante que as imagens sejam lidas na rotação correta, descartando metadados de câmeras que possam distorcer a orientação original. Nenhuma outra técnica de *Data Augmentation* (aumento de dados) foi aplicada na origem, permitindo que exploremos esses recursos nativamente com o YOLOv8 posteriormente, se necessário.

Nas seções a seguir, detalhamos as características de cada conjunto de dados:

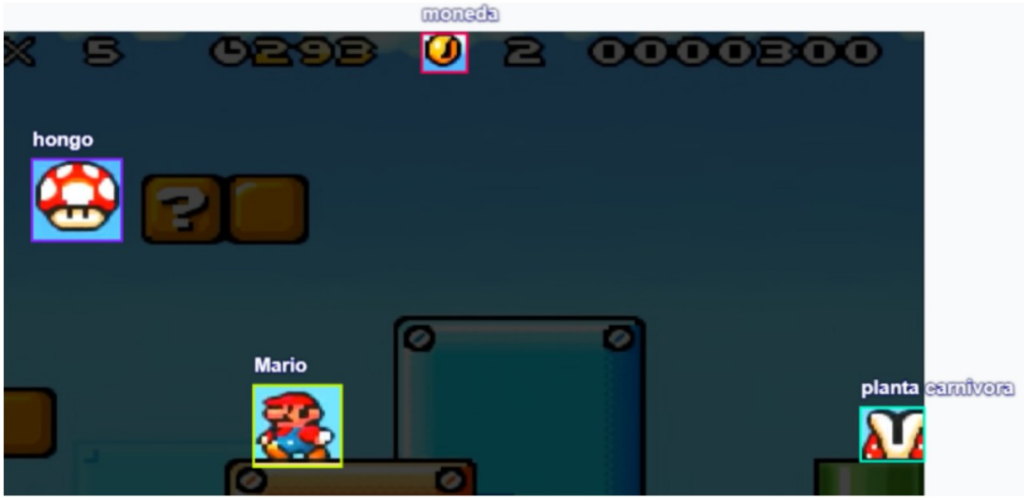
Super Mario Universe

O dataset Nelson Delgado - Mario [Delgado, 2023], contém imagens do universo dos jogos do Super Mario e é um excelente ponto de partida para o treinamento do YOLOv8. Como os objetos (personagens e itens) possuem características visuais bem definidas, cores vibrantes e bordas claras, é ideal para entendermos como a rede neural aprende a diferenciar múltiplas classes em um mesmo quadro.

Desafios e Limitações: Por conter apenas 164 imagens, este dataset apresenta um alto risco de *overfitting* (quando o modelo "decora"as imagens em vez de aprender os padrões genéricos). Será necessário monitorar as métricas de treinamento de perto e, possivelmente, aplicar técnicas de *Data Augmentation* para contornar essa limitação.

Classes Mapeadas: Os objetos que foram anotados são instâncias divididas em várias classes icônicas do jogo: *Mario, Goomba, Koopa, Moneda, Hongo (Cogumelo), Planta Carnivora, Caparazon (Casco), Hoja.* Na Figura 6.1 podemos visualizar um exemplo de imagem presente no dataset.

Figura 6.1: Exemplo de imagem dataset contendo as classes Mario,Moneda (Moeda), Hongo (Cogumelo), Planta Carnivora.



Fonte: Extraído do dataset Mario [Delgado, 2023].

Onde Está o Wally? (Find Waldo)

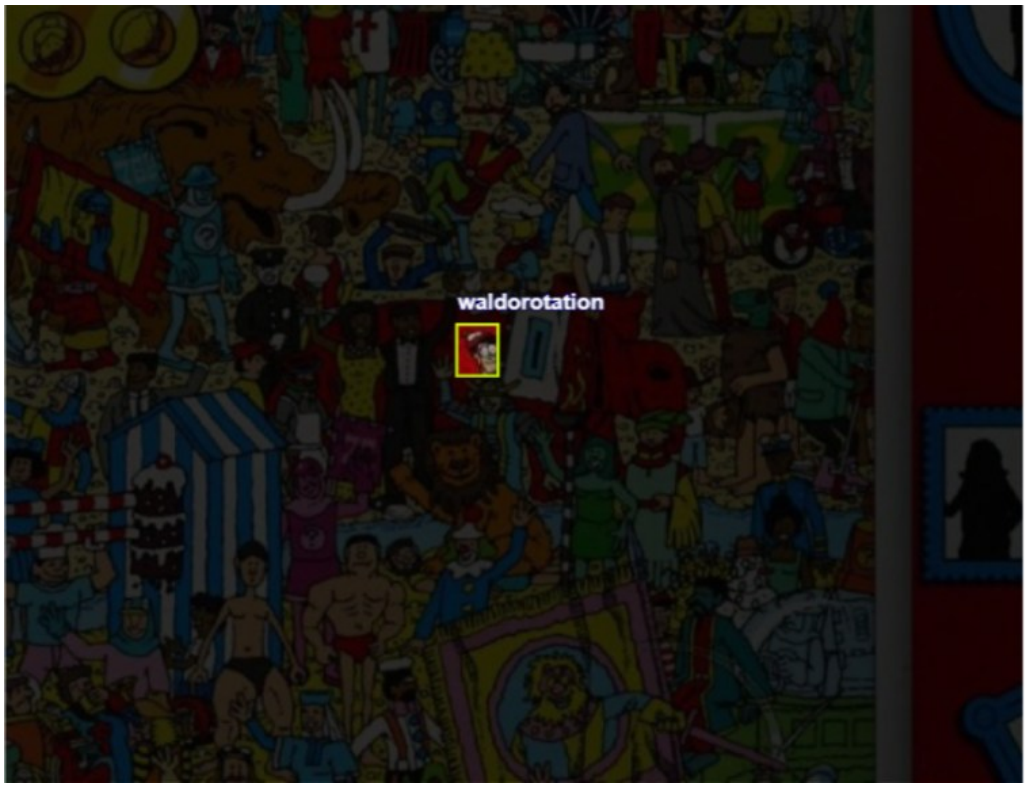
O dataset Waldowally - Waldo [Roboflow, 2024b], baseado no clássico jogo de busca visual, eleva significativamente o nível de complexidade do nosso modelo, demonstrando a capacidade do YOLOv8 de extrair características minuciosas (*features*) em fundos complexos. O desafio principal é a detecção de objetos pequenos e lidar com oclusões em cenários com altíssima poluição visual.

Desafios e Limitações: Contendo 1.154 imagens, o alvo (Wally) ocupa uma fração minúscula de cada imagem, a resolução de entrada da imagem no YOLOv8 precisa ser aumentada, o que exigirá um consumo maior de memória da GPU.

Também, temos a possibilidade de obter um alto índice de falsos positivos,os cenários são desenhados propositalmente com objetos e cores que imitam o personagem (como padrões listrados de vermelho e branco). Isso testará duramente o modelo contra detecções incorretas.

Classes Mapeadas: Nas imagens foi mapeado a localização do Wally, classe *waldo*. Na Figura 6.2 podemos visualizar um exemplo de imagem presente no dataset.

Figura 6.2: Exemplo de imagem dataset exemplificando a classe waldo.



Fonte: Extraído do dataset Waldo [Roboflow, 2024b].

Detecção de Vagas de Estacionamento (PKLot)

O *dataset PKLot - Detection of Parking Spaces* Roboflow [2024a] foca em uma aplicação real e de alto impacto no mercado: o monitoramento inteligente de estacionamentos via CFTV. O modelo aprenderá a identificar a localização das vagas e a classificá-las quanto ao seu status de ocupação sob diferentes condições de iluminação e clima. É um exemplo perfeito da aplicação do YOLOv8 em sistemas de cidades inteligentes e automação comercial.

Desafios e Limitações: O principal ponto de atenção aqui é a possibilidade de Vício de Perspectiva (Viés Geométrico). Como as imagens são geradas a partir de câmeras fixas, o modelo pode se tornar ”especialista”apenas naquele ângulo específico. Se for implementado posteriormente em uma câmera com uma angulação de visão totalmente diferente (por exemplo, mudando de uma visão lateral para uma visão aérea), a precisão do modelo poderá cair drasticamente.

Quantidade de Imagens: O *dataset* base conta com 2.766 imagens extraídas de câmeras de vigilância em dias ensolarados, nublados e chuvosos.

Classes Mapeadas: Foram marcadas nas imagens detectando vagas ocupadas e livres, classes *space-empty* (Vaga livre) e *space-occupied* (Vaga ocupada) Na Figura 6.3 podemos visualizar um exemplo de imagem presente no *dataset*.

Figura 6.3: Exemplo de imagem *dataset* exemplificando as classes *space-empty* (Vaga livre) e *space-occupied* (Vaga ocupada).

Fonte: Extraído do dataset Vagas [Roboflow, 2024a].

Divisão dos Dados (Dataset Splits)

Um passo fundamental no treinamento de redes neurais é a correta divisão dos dados para garantir que o modelo aprenda adequadamente e possa ser testado de forma justa, evitando o *overfitting* (quando o modelo ”decora”os dados em vez de generalizar os padrões).

Para manter a consistência e padronização durante o minicurso, os três datasets mencionados acima utilizam rigorosamente a mesma proporção de divisão (*split*):

- **Conjunto de Treinamento (Train) - 75%:** Esta é a maior porção dos dados. O YOLOv8 usará essas imagens e suas respectivas anotações (*bounding boxes*) para aprender os padrões visuais, ajustar seus pesos e entender como identificar cada classe.
- **Conjunto de Validação (Validation) - 15%:** Usado simultaneamente durante a fase de treinamento. A cada época (*epoch*), o modelo avalia seu desempenho provisório nestas imagens invisíveis ao treino. Isso ajuda a monitorar se o modelo está melhorando e permite ajustes nos hiperparâmetros de forma otimizada.

- **Conjunto de Teste (Test) - 10%:** Imagens mantidas isoladas até a conclusão total do treinamento. Elas simulam dados do ”mundo real”e são usadas exclusivamente no final do processo para obtermos as métricas definitivas de precisão e performance do nosso modelo.

Plataforma Roboflow: Upload, Marcação e Exportação

Na Inteligência Artificial, a precisão do modelo é diretamente proporcional à qualidade dos dados. A busca por bases de dados robustas pode ser realizada em repositórios abertos, como o Kaggle ou Roboflow Universe. Para a nossa prática inicial de marcação e anotação detalhada, utilizaremos um *dataset* público de detecção do personagem Mario [Delgado, 2023].

O **Roboflow** se destaca como ferramenta principal, permitindo anotação de *bounding boxes*, pré-processamento, aumento de dados (*data augmentation*) e exportação rápida.

Data Augmentation (Aumento de Dados)

Um passo fundamental na curadoria é o ***Data Augmentation***. Esta técnica consiste em aumentar artificialmente o tamanho e a diversidade do *dataset* através da criação de versões modificadas das imagens originais de treino.

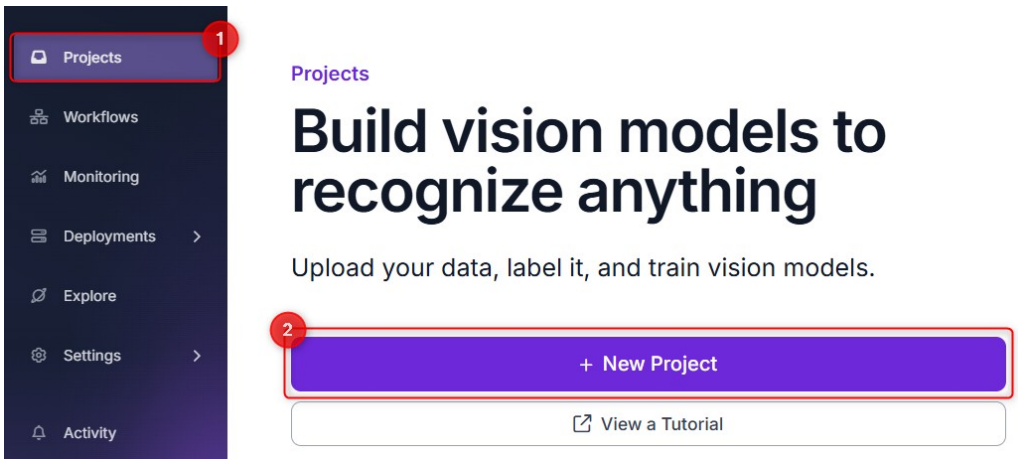
- **Principais Técnicas:** Rotação, espelhamento horizontal ou vertical (*flip*), alteração de brilho e contraste, adição de ruído e corte (*crop*).
- **Prós:** Melhora significativamente a capacidade de generalização do modelo, ajuda a prevenir o *overfitting* e é uma solução excelente quando se tem um número reduzido de imagens originais.
- **Contras:** Aumenta o tempo de treino (devido ao maior volume de dados) e, se aplicado de forma descontextualizada (ex: inverter verticalmente imagens de carros numa estrada), pode baralhar a aprendizagem do modelo.
- **Como aplicar:** O processo pode ser realizado visualmente e com facilidade diretamente na plataforma **Roboflow** (na etapa de *Generate* antes de exportar o dataset), ou via **código**, passando parâmetros de *augmentation* durante a chamada do método `train` do YOLOv8 (que já possui várias destas técnicas ativadas por padrão na sua arquitetura).

Guia Prático: Uso do Roboflows

Passo 1: Acesso a Plataforma, Criação do Projeto e Definição do Escopo

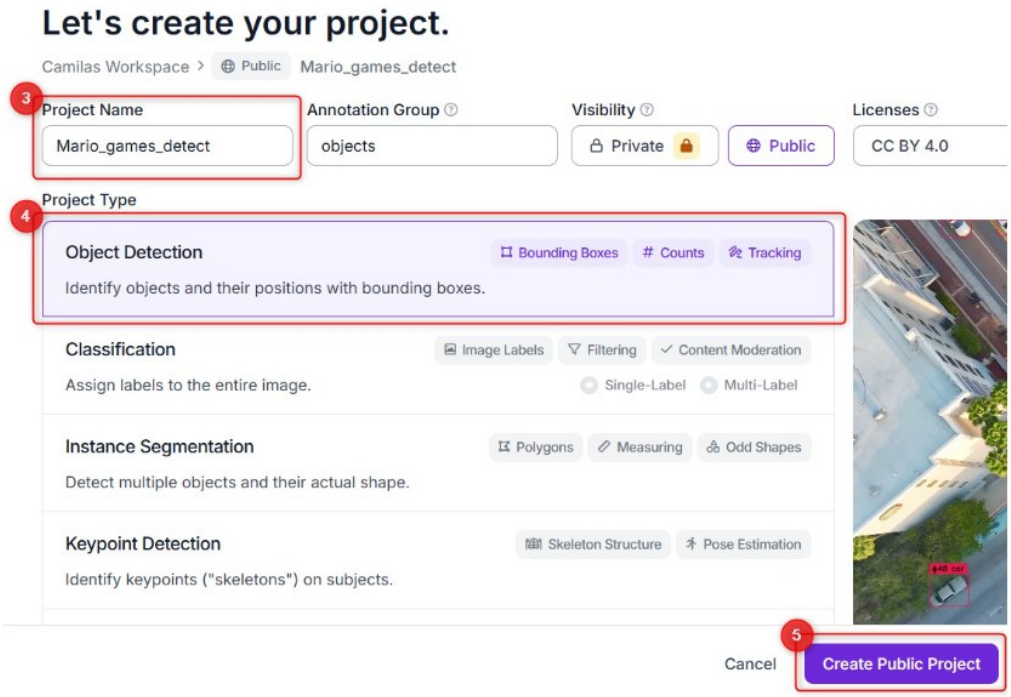
- Acesse `roboflow.com`, realize o cadastro gratuito selecionando o plano público (Public Plan) e nomeie seu espaço de trabalho.
- Na aba Projects (1), clique em + New Project (2) conforme ilustrado na Figura 6.4
- Passo 3: Preencha o nome do projeto (3) como Mario_games_detect e, obrigatoriamente, selecione o tipo de projeto como Object Detection (Bounding Boxes) (4), que instrui a ferramenta de que iremos rastrear a localização exata dos objetos e não apenas classificar a imagem inteira. Conclua clicando em Create Public Project (5), conforme ilustrado na Figura 6.5.

Figura 6.4: Início da criação do projeto.



Fonte: Elaborado pelos autores.

Figura 6.5: Interface de criação de projeto no Roboflow.

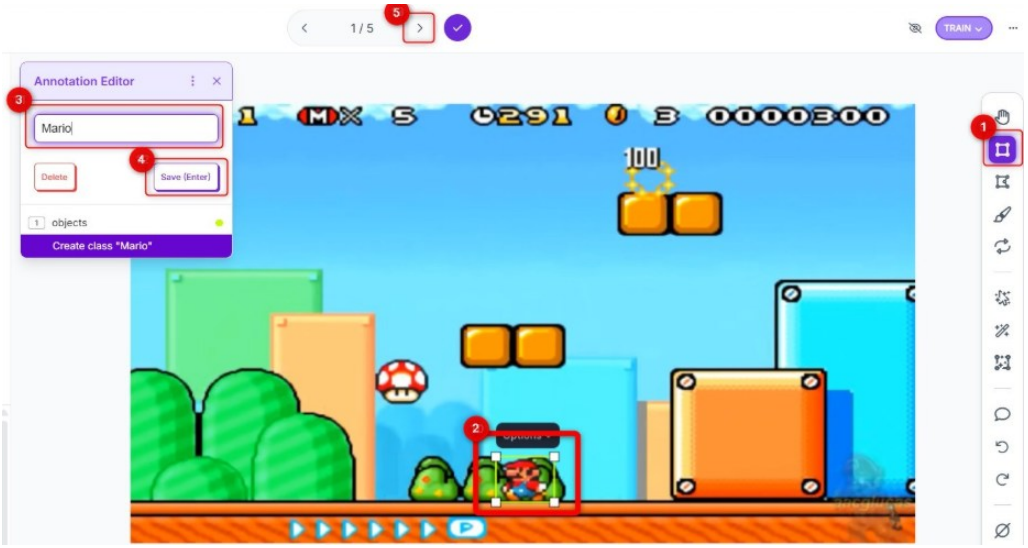


Fonte: Elaborado pelos autores.

Passo 2: Upload e Rotulagem Manual (Bounding Boxes)

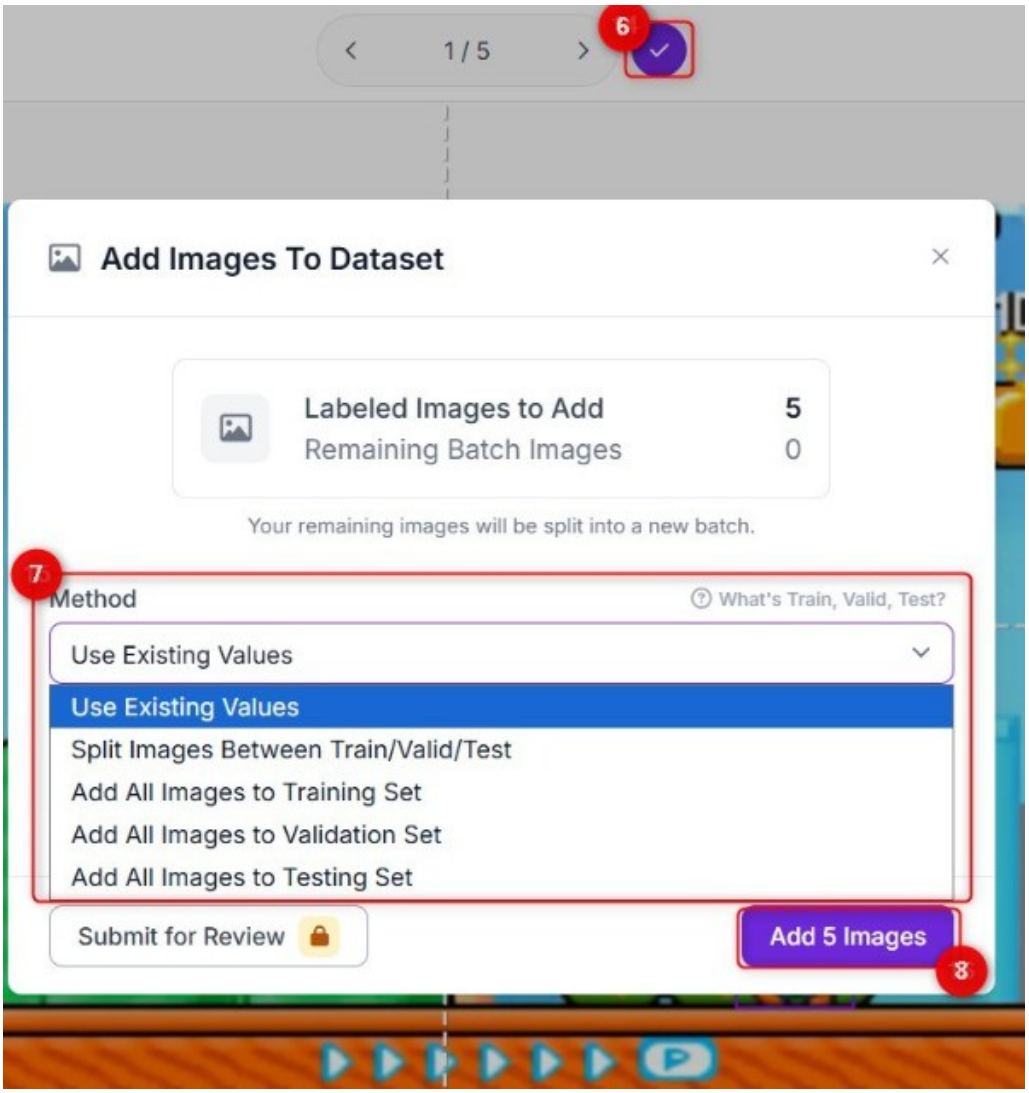
- Dentro do projeto criado, acesse a aba **Upload Data** e selecione a pasta local contendo as imagens brutas do Super Mario. Clique em **Save and Continue**.
- Vá para a aba Annotate, selecione o lote enviado e clique em Annotate Images seguido por **Label Myself** (ou Label With My Team).
- Com a ferramenta de desenho ativa (atalho no teclado ou menu lateral)(1), clique e arraste sobre o objeto de interesse na imagem (ex: o personagem Mario) para criar o retângulo de marcação(2). Digite o nome correspondente da classe no editor flutuante(3) e salve(4). Use a seta superior para navegar entre as imagens(5), conforme ilustrado na Figura 6.6.
- Ao finalizar o lote, clique no ícone de confirmação no topo da tela(6). Na janela pop-up, defina o método como **Split Images Between Train/Valid/Test**(7) para que a plataforma distribua automaticamente as imagens marcadas nas proporções corretas do curso por fim adicione as imagens selecionando o botão add X imagens (8), conforme ilustrado na Figura 6.7.

Figura 6.6: Aplicação da Bounding Box sobre o objeto desejado definindo sua classe.



Fonte: Elaborado pelos autores.

Figura 6.7: Processo de incorporação de imagens anotadas ao dataset.

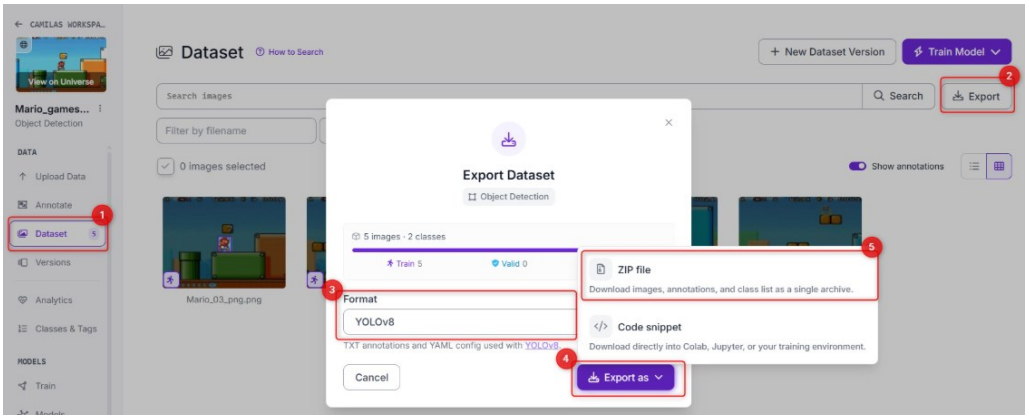


Fonte: Elaborado pelos autores.

Passo 3: Configuração da Versão e Exportação para o YOLOv8

- Vá para a aba **Versions** (ou acesse através do menu lateral da aba Dataset) (1) e adicione a etapa de pré-processamento de Auto-orientação.
- Clique em **Generate** para processar e consolidar a primeira versão oficial do seu dataset.
- Com a versão gerada, clique no botão **Export**(2) no canto superior direito.
- No campo **Format**(3), selecione rigorosamente a opção **YOLOv8**. Escolha o método de download como **ZIP file** (para download direto da pasta compactada contendo as imagens e as marcações em formato .txt)(5) ou selecione **Show Code Snippet** para obter a chave de API que automatiza o download via código Python diretamente no Jupyter Notebook, conforme ilustrado na Figura 6.8.

Figura 6.8: Exportar Dataset - Download do arquivo ZIP.



Fonte: Elaborado pelos autores.

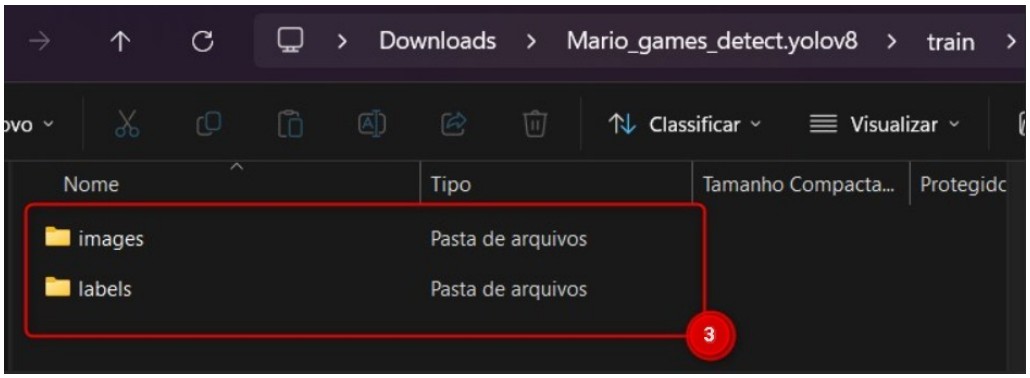
Passo 4: Estrutura do Dataset Exportado (O que há no ZIP?)

Ao optar pelo download via arquivo ZIP e descompactá-lo na sua máquina local, você encontrará uma estrutura de diretórios rigorosamente formatada para o padrão exigido pelo YOLOv8, conforme ilustrado na Figura 6.10. É fundamental entender como os dados estão organizados antes de iniciar o código:

- Arquivos na Raiz do Diretório:
  - **data.yaml**: É o arquivo de configuração principal. Ele funciona como um mapa, indicando ao algoritmo do YOLO os caminhos exatos das pastas e a lista de nomes das classes que ele deve aprender.
  - **README.roboflow:** Documento de texto contendo os metadados da exportação, link original e informações de licença do dataset.

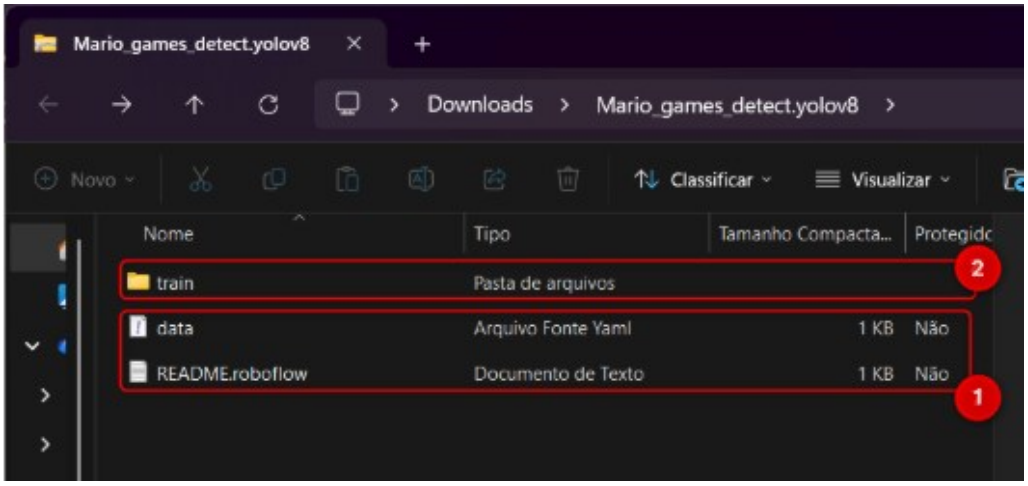
- **Pastas de Divisão (Splits)**: A raiz também conterá três pastas correspondentes à divisão explicada anteriormente: *train* (treino), *valid* (validação) e *test* (teste).
- **Subpastas Internas (Imagens e Rótulos)**: Para que o YOLOv8 funcione, as imagens e as marcações não ficam juntas na mesma pasta. Ao abrir qualquer uma das pastas de divisão (como a *train*, por exemplo), você encontrará sempre duas subpastas paralelas, conforme ilustrado na Figura 6.9:
  - A pasta **images/**: Contém exclusivamente os arquivos visuais (as fotografias em formato *.jpg* ou *.png*).
  - A pasta **labels/**: Contém arquivos de texto (*.txt*) com exatamente o mesmo nome das imagens correspondentes. Dentro de cada arquivo *.txt* estão as coordenadas matemáticas numéricas (posição X, posição Y, largura e altura) referentes à bounding box desenhada sobre a imagem.

Figura 6.9: Exemplo das Subpastas */train/labels/* e */train/images/* do dataset Delgado [2023].



Fonte: Elaborado pelos autores.

Figura 6.10: Exemplo da exportação em formato ZIP do dataset Delgado [2023].



Fonte: Elaborado pelos autores.

6.7 Treinamento no Jupyter Notebook: Desenvolvimento do Algoritmo

As seções a seguir demonstram o fluxo de execução organizado por células, comum em ambientes interativos como o Jupyter Notebook. O arquivo fonte principal (`codigo.py`) é carregado fragmentado para acompanhamento didático.

Célula 1: Instalação de Dependências

O primeiro passo, ilustrado no Código-fonte 74, consiste em instalar as bibliotecas base. No caso de dispositivos com placa de vídeo NVIDIA, importa-se também as rotinas CUDA (`cu121`) para viabilizar a aceleração por hardware.

Código-fonte 74 Célula 1 - Instalação de pacotes e CUDA.

```
# CÉLULA 1: Instalação
!pip install ultralytics roboflow -q
!pip install torch torchvision torchaudio --index-url
↪  https://download.pytorch.org/whl/cu121 #Apenas caso utilize GPU da
↪  NVIDIA
```

Fonte: Elaborado pelos autores.

Célula 2: Verificação do Hardware (Em caso de GPU NVIDIA)

Antes de iniciar processos pesados, o código-fonte 75 realiza uma verificação para garantir que o *framework* comunica perfeitamente com a GPU.

Código-fonte 75 Célula 2 - Verificação de Hardware de vídeo.

```
# CÉLULA 2: Verificação da Placa de Vídeo (Caso utilize)
import torch
print("A GPU está ativada?", torch.cuda.is_available())
```

Fonte: Elaborado pelos autores.

Célula 3: Extração do Dataset

A aquisição do dataset pode ser feita de diversas formas. Estruturamos abaixo dois métodos comuns. O participante deve escolher apenas um dos métodos para prosseguir. Ao final de ambos, a variável `caminho_yaml` ou a própria pasta garantirá a referência correta.

Método 1: Upload Manual via Arquivo .ZIP No código-fonte 76, abordamos a descompactação caso o participante tenha feito o download do ZIP.

Código-fonte 76 Célula 3 (Método 1) - Descompactação via ZIP.

```
# CÉLULA 3: Extração do Dataset
# METODO 1: Upload Manual via Arquivo .ZIP
import zipfile
import os

caminho_zip = rf"INFORME AQUI O CAMINHO DO DIRETÓRIO ONDE ESTA SALVO O DATASET"
pasta_destino = "INFORME AQUI O NOME DA PASTA ONDE FICARÁ SALVO O DATASET DESCOMPACTADO"

if os.path.exists(caminho_zip):
    with zipfile.ZipFile(caminho_zip, 'r') as zip_ref:
        zip_ref.extractall(pasta_destino)
    print("Dataset descompactado com sucesso!")
```

Fonte: Elaborado pelos autores.

Método 2: Aquisição Automatizada via API do Roboflow O código-fonte 77 é o método mais recomendado por automatizar a transferência utilizando as credenciais da API do Roboflow.

Código-fonte 77 Célula 3 (Método 2) - Integração via API do Roboflow.

```
# METODO 2: Download Automatizado via API do Roboflow
#Substitua as credenciais pelas do seu projeto (aba "Export" noRoboflow)
from roboflow import Roboflow
rf = Roboflow(api_key="SUA_CHAVE_DE_API_AQUI")
project= rf.workspace("nome-do-seu-workspace").project("nome-do-seu-projeto")
version = project.version(1)
dataset = version.download("yolov8")
caminho_yaml = f"{dataset.location}/data.yaml" #Referencia para o treinamento
```

Fonte: Elaborado pelos autores.

Célula 4: Treinamento e Monitoramento de Tempo

É inciado o treinamento da rede neural, ilustrado no código-fonte 78. O modelo base é instanciado e a biblioteca `time` avalia a duração do ciclo. O parâmetro `verbose=False` suprime a poluição visual no terminal.

Código-fonte 78 Célula 4 - Treinamento da rede neural com cronometragem.

```
# CÉLULA 4: Treinamento com Cronômetro
from ultralytics import YOLO
import time

model = YOLO('yolov8n.pt')
caminho_yaml = f"{pasta_destino}/data.yaml"

print("Iniciando o treinamento...")
tempo_inicio = time.time()

results = model.train(
    data=caminho_yaml,
    epochs=300,
    imgsz=512,
    plots=True,
    verbose=False  # Oculta poluição visual no terminal
)

tempo_fim = time.time()
tempo_total_segundos = tempo_fim - tempo_inicio
minutos = int(tempo_total_segundos // 60)
segundos = int(tempo_total_segundos % 60)

print(f"Treinamento concluído em {minutos} minutos e {segundos}
↪  segundos.")
```

Fonte: Elaborado pelos autores.

Comparação de Desempenho

Consolidamos o tempo gasto processando datasets variados em diferentes hardwares, As Tabelas 6.3, 6.4 e 6.5 detalham os desempenhos individuais por dataset.

Tabela 6.3: Comparativo de Tempo de Treinamento - Dataset Waldo.

Hardware	Épocas	Tempo/Época	Tempo Total
Nitro V15 (GPU)	5	9,5	47,5s
Lab. Faculdade (CPU)	5	132s	11m 02s
Google Colab (GPU)	5	17,8	1m 29s

Fonte: Elaborado pelos autores.

Tabela 6.4: Comparativo de Tempo de Treinamento - Dataset Vagas/PKLot.

Hardware	Épocas	Tempo/Época	Tempo Total
Nitro V15 (GPU)	5	31,9s	2m 39s
Lab. Faculdade (CPU)	5	411s	34m 15s
Google Colab (GPU)	5	158,8s	13m 14s

Fonte: Elaborado pelos autores.

Tabela 6.5: Comparativo de Tempo de Treinamento - Dataset Mario.

Hardware	Épocas	Tempo/Época	Tempo Total
Nitro V15 (GPU)	5	1,89s	0m 9,48s
Lab. Faculdade (CPU)	5	470s	1m 34s
Google Colab (GPU)	5	6,2s	0m 31s

Fonte: Elaborado pelos autores.

Diagnóstico de Modelo: Overfitting, Underfitting e Ponto Ideal

Durante o treinamento de uma rede neural, buscamos o equilíbrio entre a capacidade do modelo de aprender o padrão dos dados e sua habilidade de generalizar para novos cenários. Três estados principais podem ser identificados:

- ***Overfitting* (Sobreadaptação):** Ocorre quando o modelo ”decora”as imagens de treino, apresentando um falso sucesso, mas falhando miseravelmente ao processar imagens novas na vida real. A curva de erro de treino continua a descer, contudo a curva de erro de validação para de descer e começa a subir.
- ***Underfitting* (Subadaptação):** O modelo não possui complexidade ou treino suficiente para aprender os padrões básicos. As métricas de erro mantêm-se altas desde a primeira época e a precisão não sobe.
- **Ponto Ideal (*Desirid Fit*):** É o estágio onde o modelo minimiza o erro tanto no treino quanto na validação, alcançando a máxima performance de generalização.

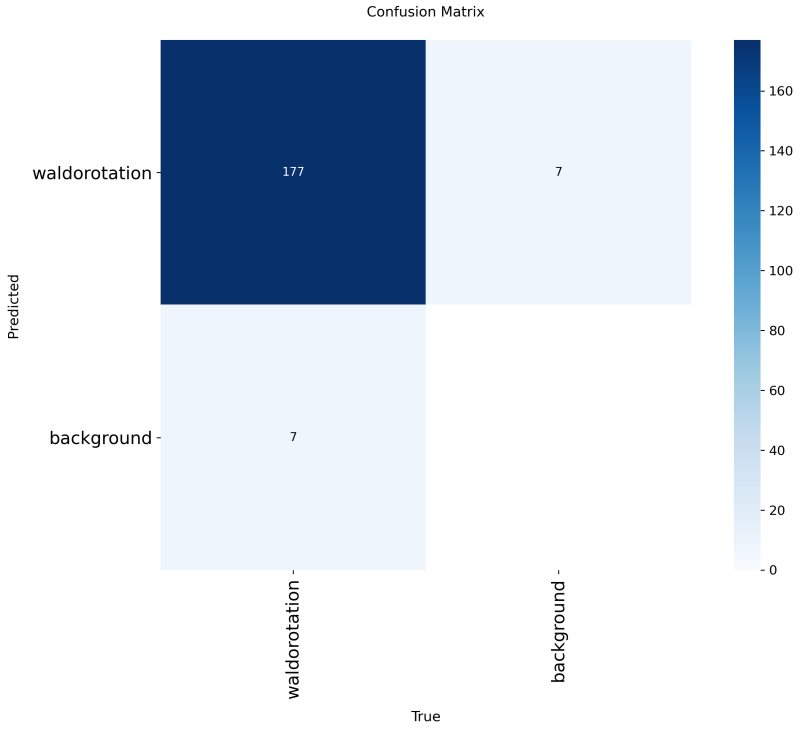
Célula 5: Validação e Análise de Métricas Finais

Após o término do treinamento, avalia-se a generalização e qualidade do modelo. Carregando os pesos otimizados (`best.pt`) utilizando o código-fonte 79, a IA gera gráficos de resultados, a Matriz de Confusão e extrai métricas como:

- **Precision (Precisão):** Mede a assertividade do acerto, indicando a taxa de falsos positivos (alarmes falsos).
- **Recall (Revocação):** Mede a capacidade do modelo de cobrir todas as ocorrências reais, evitando que objetos passem despercebidos.
- **mAP (Mean Average Precision):** Representa a acurácia macro do modelo para detectar e desenhar corretamente a *bounding box*.
- **Como ler a Matriz de Confusão (*confusion_matrix.png*):** Este gráfico possui os valores reais (verdadeiros) no eixo inferior e os valores previstos pelo

modelo no eixo lateral. A **diagonal principal** exibe os acertos absolutos. Valores fora desta diagonal representam erros. Por exemplo, se na coluna da classe pretendida e na linha indicativa de "background"(fundo) existir um número elevado, significa que o modelo está a assinalar objetos onde na verdade apenas existe cenário vazio (falsos positivos). Podemos visualizar na Imagem 6.11 um exemplo de uma Matriz de Confusão gerada a partir do treinamento efetuado durante o minicurso para o dataset Roboflow [2024b]

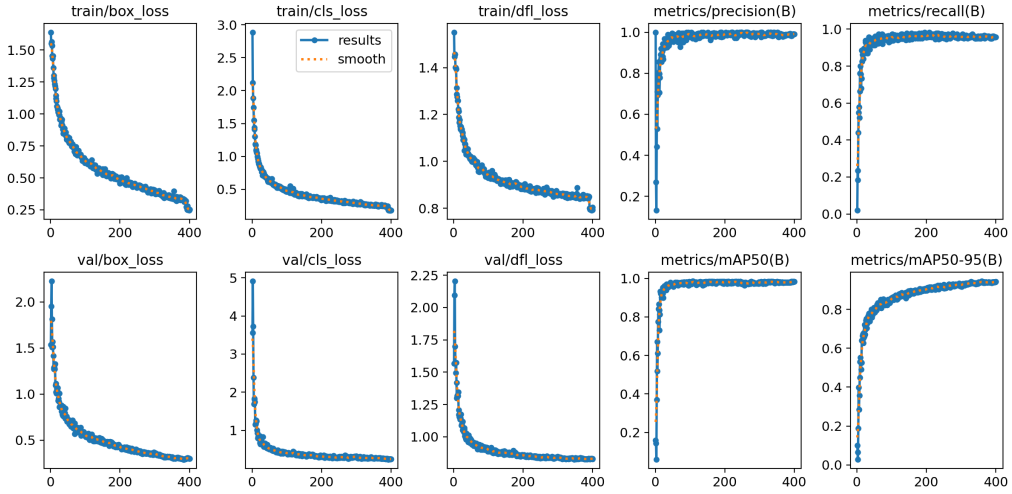
Figura 6.11: Exemplo de Matriz de Confusão gereada a partir do treinamento efetuado com o dataset Roboflow [2024b] em 400 épocas sem aumento de dados.



Fonte: Elaborado pelos autores.

- **Como ler o Gráfico de Resultados (*results.png*):** O YOLO consolida a evolução de todo o treino num único painel. As três colunas da esquerda detalham as perdas (*losses*) de treino e validação, devendo apresentar uma **tendência contínua de descida**. As colunas da direita detalham as métricas (*Precision*, *Recall* e *mAP*), devendo estas formar uma **curva ascendente que estabiliza** de forma quase plana no topo do gráfico. Podemos visualizar na Imagem 6.12 um exemplo de um gráfico de resultados gerado a partir do treinamento efetuado durante o minicurso para o dataset Roboflow [2024b]

Figura 6.12: Exemplo de gráfio *results.png* gerado a patir do treinamento efetuado com o dataset Roboflow [2024b] em 400 épocas sem aumento de dados.



Fonte: Elaborado pelos autores.

Código-fonte 79 Célula 5 - Apuração de Métricas e Geração de Gráficos.

```
58 # CÉLULA 5: Validação, Métricas e Matriz de Confusão
59 from IPython.display import display, Image
60
61 print("GRÁFICOS DO TREINAMENTO:")
62 if os.path.exists("runs/detect/train/results.png"):
63     display(Image(filename="runs/detect/train/results.png", width=800))
64     display(Image(filename="runs/detect/train/ confusion_matrix.png",
       ↪  width=600))
65
66 # Puxa o cérebro treinado da pasta padrão 'runs'
67 modelo_treinado = YOLO("runs/detect/train/weights/best.pt")
68 metricas = modelo_treinado.val(verbose=False)
69
70 print("\n" + "="*60)
71 print(f"Precisão (Evita alarmes falsos): {metricas.box.mp:.4f}")
72 print(f"Recall (Evita deixar alvo escapar): {metricas.box.mr:.4f}")
73 print(f"Acurácia Geral (mAP@50): {metricas.box.map50:.4f}")
74 print(f"Acurácia Rigorosa (mAP@50-95): {metricas.box.map:.4f}")
75 print("="*60)
```

Fonte: Elaborado pelos autores.

Ao final do treinamento em 400 épocas, é extraindo as métricas compiladas na Tabela 6.6 demostrando a qualidade analítica final de cada IA.

Tabela 6.6: Apresentação das Métricas Finais de Treinamento (400 Épocas).

Dataset	Precisão	Recall	mAP@50	mAP@50-95	Tempo Gasto
Waldo (Wally)	98,31%	96,20%	98,09%	94,34%	63m 49s
Vagas (PKLot)	98,23%	98,11%	98,98%	94,71%	213m 7s
Mario (Mario)	79,51%	96,48%	95,98%	62,63%	12m 39s

Fonte: Elaborado pelos autores.

Célula 6: Teste Visual (Inferência)

Para concluir o ciclo, o código-fonte 80 aplica a rede neural treinada em uma imagem inédita, retornando as marcações visuais na foto.

Código-fonte 80 Célula 6 - Inferência do modelo com predição visual.

```
# CÉLULA 6: Predição Final e Plotagem
imagem_teste = rf"INFORME AQUI O DIRETÓRIO DA IMAGEM PARA VALIDAÇÃO"

print("Procurando o objeto...")
resultados_pred = modelo_treinado.predict(
    source=imagem_teste,
    conf=0.5,
    save=True
)

pasta_salva = resultados_pred[0].save_dir
nome_arquivo = os.path.basename(resultados_pred[0].path)
caminho_imagem_final = os.path.join(pasta_salva, nome_arquivo)

print("RESULTADO VISUAL:")
display(Image(filename=caminho_imagem_final, width=800))
```

Fonte: Elaborado pelos autores.

6.8 Declaração de uso de IA generativa

☐ NÃO utilizei ferramentas de IA generativa para produzir, reescrever ou expandir trechos do texto do meu mini-curso. Eventuais correções automáticas de ortografia/gramática do editor de texto não se enquadram nesta declaração.

☑ UTILIZEI ferramentas de IA generativa de forma ética e transparente, exclusivamente como apoio redacional (ex.: revisão gramatical, melhoria de clareza, sugestão de organização), sem terceirização do conteúdo intelectual. Declaro que: (a) informei o uso; (b) revisei integralmente e sou responsável(a) pelo texto final; (c) conferi a veracidade de dados, citações e referências, garan-

tindo que não há referências inventadas, trechos sem fonte ou conteúdo não verificável.

Se marcou acima a opção UTILIZEI, descreva de forma objetiva:

- **Ferramenta(s)/versão(ões):** Gemini 3.1 Pro;
- **Seções/trechos impactados:** Estrutura total do documento impactada;
- **Finalidade(s):** Estruturação de tópicos para melhorar a clareza e fluxo do conteúdo e correção de erros no LaTeX.

6.9 Bibliografia

Delgado, N. (2023). Mario dataset. https://universe.roboflow.com/nelson-delgado/mario. Acesso em: 17 maio 2026.

Foundation, P. S. (2026). venv — criação de ambientes virtuais. https://docs.python.org/pt-br/3/library/venv.html. Acesso em: 17 maio 2026.

Roboflow (2024a). Detection of parking spaces. https://universe.roboflow.com/pklot/detection-of-parking-spaces. Acesso em: 17 maio 2026.

Roboflow (2024b). Waldo qa5u7 dataset. https://universe.roboflow.com/waldowally/waldo-qa5u7. Acesso em: 17 maio 2026.

Sohan, M., Sai Ram, T., and Rami Reddy, C. V. (2024). A review on yolov8 and its advancements. In Jacob, I. J., Piramuthu, S., and Falkowski-Gilski, P., editors, *Data Intelligence and Cognitive Informatics*, pages 529–545, Singapore. Springer Nature Singapore.

Ultralytics (2026a). Documentação oficial do yolov8. https://docs.ultralytics.com/pt/models/yolov8. Acesso em: 17 maio 2026.

Ultralytics (2026b). Yolov8: State-of-the-art object detection. https://yolov8.com/. Acesso em: 17 maio 2026.

Minicurso 7

Aprenda C++ na prática: Desenvolvendo o jogo Connect4

Paulo Felipe Candia Cazon e Sidiney Barbosa de Souza

Resumo

Este artigo descreve a estrutura de um minicurso introdutório de 90 minutos sobre programação em C++, voltado para iniciantes absolutos. A abordagem prática utiliza o desenvolvimento do clássico jogo Connect 4 como fio condutor do aprendizado, cobrindo conceitos fundamentais como variáveis primitivas, matrizes bidimensionais, estruturas de controle e funções. Uma interface gráfica é construída com a biblioteca SDL2, permitindo que os alunos vejam o resultado direto do seu código em uma aplicação visual e interativa.

7.1 Introdução

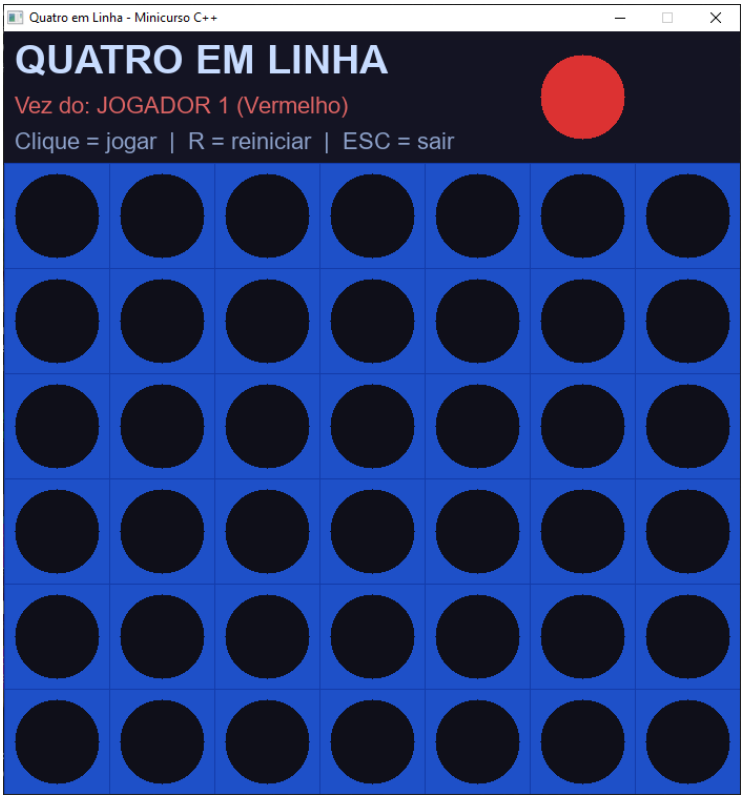
A linguagem C++ é uma das mais utilizadas no mundo da programação, presente em sistemas operacionais, jogos, aplicações de alto desempenho e softwares embarcados. Apesar da sua relevância, é frequentemente considerada difícil para iniciantes devido à sua sintaxe mais rígida e à necessidade de compreender conceitos como gerenciamento de memória e tipagem estática.

Este minicurso propõe uma abordagem diferente: ao invés de ensinar C++ de forma isolada e abstrata, utilizamos o desenvolvimento de um jogo real — o Connect 4 (Quatro em Linha) — como fio condutor do aprendizado. Dessa forma, cada conceito ensinado tem uma aplicação imediata e visível, tornando o processo mais motivador e concreto.

O Connect 4 é um jogo de tabuleiro para dois jogadores, no qual cada um se alterna colocando peças em um tabuleiro de 7 colunas e 6 linhas. As peças caem pela gravidade até a posição mais baixa disponível na coluna escolhida. Vence quem

alinhar 4 peças consecutivas na horizontal, vertical ou diagonal [Hasbro, 2014]. A Figura 7.1 ilustra um estado típico de partida.

Figura 7.1: Tela do jogo Connect 4 desenvolvido no minicurso.



Fonte: Autoria própria.

A simplicidade das regras do jogo permite que o foco esteja nos conceitos de programação, e não na complexidade do domínio do problema. Ao longo do minicurso, os alunos serão guiados desde a configuração do ambiente de desenvolvimento até a execução de um jogo funcional com interface gráfica, utilizando a biblioteca SDL2 para renderização visual.

7.2 Público-alvo

O minicurso é destinado a pessoas com conhecimento básico em informática — ou seja, que saibam utilizar um computador, navegar pela internet e instalar programas — mas que nunca tenham tido contato com programação ou lógica computacional.

Não é necessário nenhum conhecimento prévio de linguagens de programação. O conteúdo foi estruturado para que qualquer pessoa com curiosidade e vontade de aprender possa acompanhar todas as etapas dentro dos 90 minutos disponíveis.

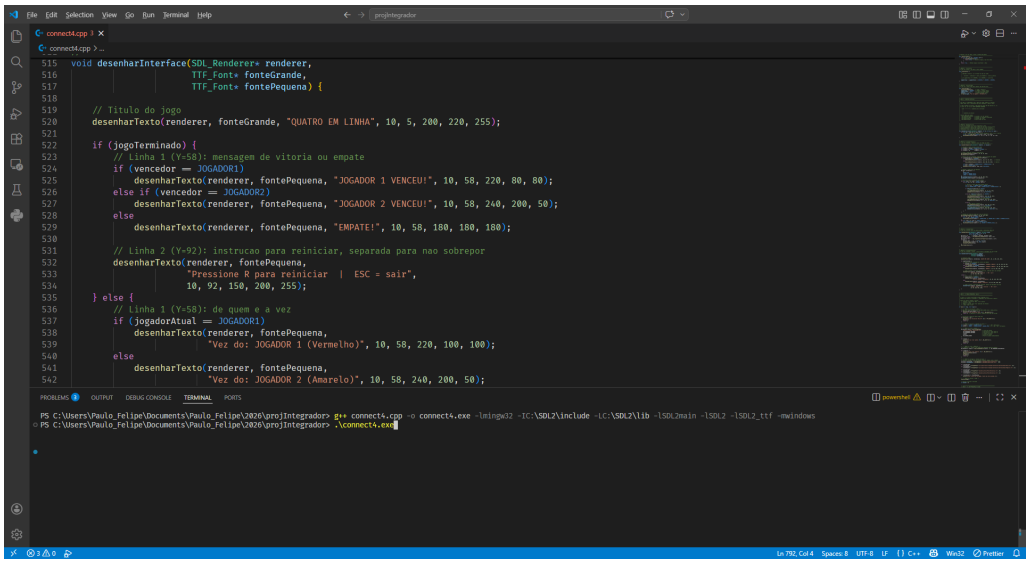
7.3 Pré-requisitos

Para participar do minicurso, os alunos precisarão ter instalado previamente em seus computadores:

- **Sistema operacional:** Windows 10 ou superior ou algum distro do Linux (Ubuntu por exemplo) ;
- **Visual Studio Code (VS Code):** editor de código gratuito, disponível em Microsoft [2026];
- **MinGW-w64:** compilador C++ para Windows;
- **Biblioteca SDL2** e **SDL2_ttf:** bibliotecas gráficas para criação da janela e renderização de texto [Community, 2025].

Um guia de instalação passo a passo será disponibilizado antes do evento, para que os alunos cheguem com o ambiente já configurado. A Figura 7.2 mostra o ambiente de desenvolvimento que será utilizado.

Figura 7.2: Ambiente de desenvolvimento: VS Code com o arquivo connect4.cpp e terminal integrado.



Fonte: Autoria própria.

7.4 Introdução à Biblioteca SDL2

A SDL2 (*Simple DirectMedia Layer*) é uma biblioteca gratuita, de código aberto e multiplataforma que fornece acesso a hardware gráfico, de áudio e de entrada (teclado, mouse, controle) por meio de uma interface simples em C. É amplamente utilizada no desenvolvimento de jogos e aplicações multimídia, sendo a base de jogos comerciais como o próprio motor do Valve.

Para que um programa com SDL2 funcione, três elementos são indispensáveis: a **inicialização** da biblioteca, a criação de uma **janela** (Figura 7.3) para exibir o conteúdo, e a criação de um **renderer**, que é o objeto responsável por todos os desenhos. Ao final do programa, todos esses recursos precisam ser liberados explicitamente.

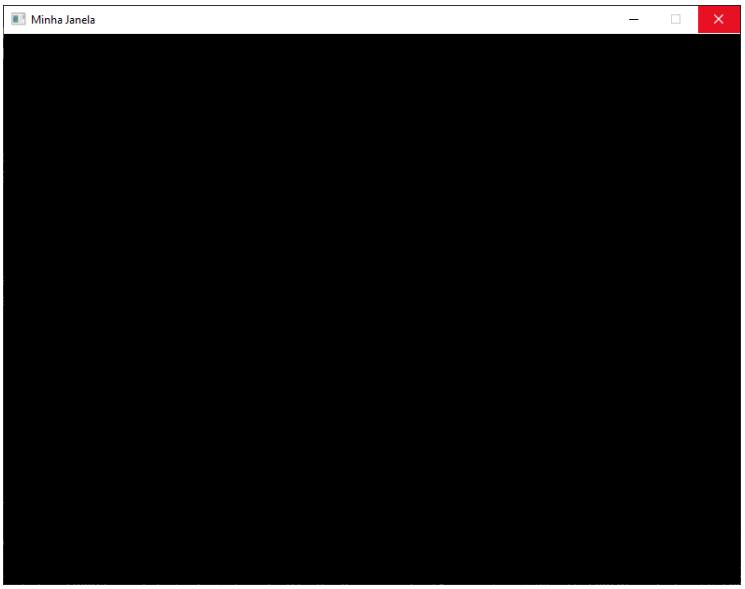
A estrutura mínima de qualquer programa SDL2 segue sempre o esqueleto presente no Algoritmo 81:

Código-fonte 81 Algoritmo com a estrutura base do SDL.

```
#include <SDL2/SDL.h>

int main(int argc, char* argv[]) {
    SDL_Init(SDL_INIT_VIDEO);

    SDL_Window* janela = SDL_CreateWindow(
        "Minha Janela",
        SDL_WINDOWPOS_CENTERED,
        SDL_WINDOWPOS_CENTERED,
        800, 600,
        SDL_WINDOW_SHOWN
    );

    SDL_Renderer* renderer = SDL_CreateRenderer(
        janela, -1, SDL_RENDERER_ACCELERATED
    );

    SDL_Delay(10000);

    SDL_DestroyRenderer(renderer);
    SDL_DestroyWindow(janela);
    SDL_Quit();
    return 0;
}
```

Fonte: Autoria própria.

Figura 7.3: Janela do SDL resultado do Algoritmo 81.



Fonte: Autoria própria.

`SDL_Init(SDL_INIT_VIDEO)` ativa o subsistema gráfico da biblioteca. `SDL_CreateWindow` cria a janela com título, posição, largura, altura e flags de exibi-

ção. `SDL_CreateRenderer` cria o renderer vinculado à janela; o segundo argumento `-1` instrui a SDL2 a escolher automaticamente o melhor driver disponível. Ao final, `SDL_DestroyRenderer`, `SDL_DestroyWindow` e `SDL_Quit` liberam todos os recursos alocados.

O sistema de coordenadas da SDL2 posiciona a origem $(0, 0)$ no **canto superior esquerdo** da janela. O eixo x cresce para a direita e o eixo y cresce para **baixo**, o que é diferente do plano cartesiano convencional e deve ser considerado ao posicionar qualquer elemento na tela.

Antes de desenhar qualquer coisa, é necessário definir a cor do "pincel" com `SDL_SetRenderDrawColor`, que recebe quatro valores entre 0 e 255: vermelho (R), verde (G), azul (B) e transparência (A). Após todos os desenhos do frame, `SDL_RenderPresent` exibe o resultado na tela, e `SDL_RenderClear` limpa o frame anterior para começar o próximo.

Desenhando uma Linha

O elemento mais simples que a SDL2 permite desenhar é uma linha reta entre dois pontos. A função `SDL_RenderDrawLine` recebe as coordenadas (x_1, y_1) do ponto inicial e (x_2, y_2) do ponto final.

Adicionando as seguintes linhas presentes no Algoritmo 82 entre a parte do `SDL_Renderer` e do `SLD_DestroyRendered` é possível desenhar uma linha branca diagonal do canto superior esquerdo até o canto inferior direito, mantendo-a visível por 5 segundos, mostrado na Figura 7.4:

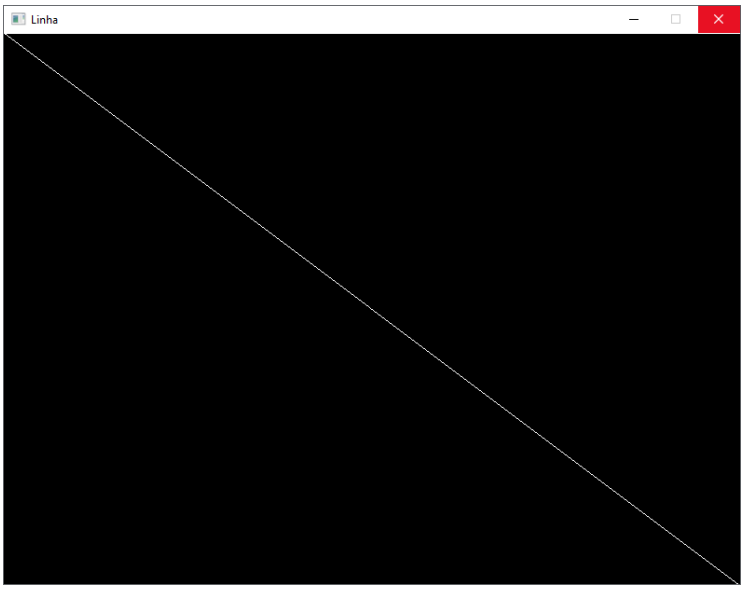
Código-fonte 82 Algoritmo para desenhar uma linha dentro do SDL.

```
1         SDL_SetRenderDrawColor(renderer, 255, 255, 255, 255);
2
3         SDL_RenderDrawLine(renderer, 0, 0, 800, 600);
4
5         SDL_RenderPresent(renderer);
6         SDL_Delay(5000);
```

Fonte: Autoria própria.

A sequência de chamadas segue sempre a mesma ordem: escolher a cor com `SDL_SetRenderDrawColor`, executar o desenho, e depois chamar `SDL_RenderPresent` para tornar o resultado visível. Qualquer desenho feito antes de `SDL_RenderPresent` fica em um buffer interno e não aparece na tela até que essa função seja chamada.

Figura 7.4: Janela do SDL mostrando a linha diagonal.



Fonte: Autoria própria.

Desenhando um Retângulo

Para desenhar um retângulo (Figura 7.5), a SDL2 utiliza a estrutura `SDL_Rect`, que agrupa quatro valores inteiros: `x` e `y` (coordenadas do canto superior esquerdo), `w` (largura) e `h` (altura), todos em pixels.

Existem duas funções para retângulos: `SDL_RenderDrawRect` desenha apenas o contorno, enquanto `SDL_RenderFillRect` preenche o interior com a cor atual. O Algoritmo 83 desenha um retângulo azul preenchido no centro da janela:

Código-fonte 83 Algoritmo para desenhar um quadrado dentro do SDL.

```
1         SDL_Rect retangulo = {300, 200, 200, 200};
2
3         SDL_SetRenderDrawColor(renderer, 30, 80, 200, 255);
4         SDL_RenderFillRect(renderer, &retangulo);
5
6         SDL_RenderPresent(renderer);
7         SDL_Delay(5000);
8
```

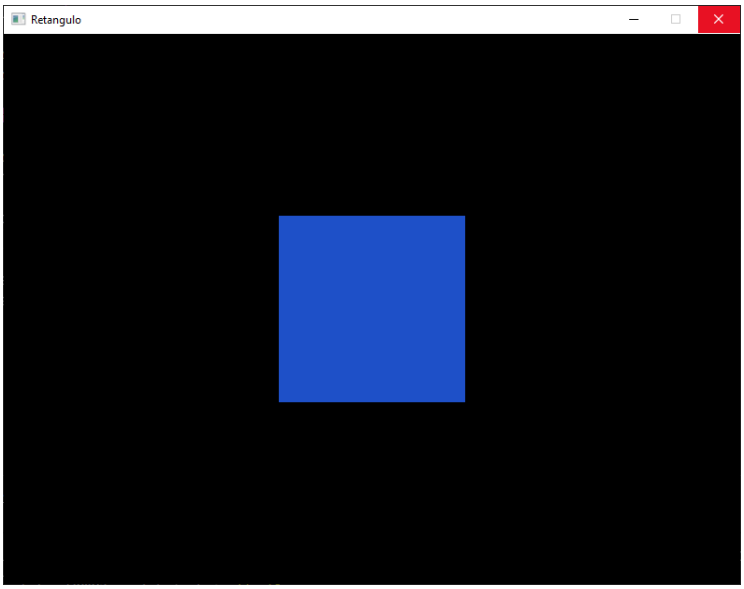
Fonte: Autoria própria.

A estrutura `SDL_Rect retangulo = {300, 200, 200, 200}` define um quadrado de 200×200 pixels cujo canto superior esquerdo está na posição $(300, 200)$. O operador `&` antes de `retangulo` passa o endereço da estrutura para a função, que é o comportamento esperado pela API da SDL2.

Desenhando um Círculo

A SDL2 não possui uma função nativa para desenhar círculos. A solução é construí-los manualmente, aproveitando o Teorema de Pitágoras: em um círculo de raio r cen-

Figura 7.5: Janela SDL mostrando o quadrado.



Fonte: Autoria própria.

trado em (cx, cy), para cada deslocamento vertical dy entre $-r$ e r, o deslocamento horizontal correspondente é $dx = \sqrt{r^2 - dy^2}$, como mostrado no Algoritmo 84.

Isso significa que podemos desenhar o círculo como uma sequência de linhas horizontais, uma para cada valor de dy. Cada linha vai de $(cx - dx, cy + dy)$ até $(cx + dx, cy + dy)$. O conjunto dessas linhas, empilhadas verticalmente, forma um disco preenchido mostrado na Figura 7.6:

Código-fonte 84 Algoritmo para desenhar um circulo dentro do SDL.

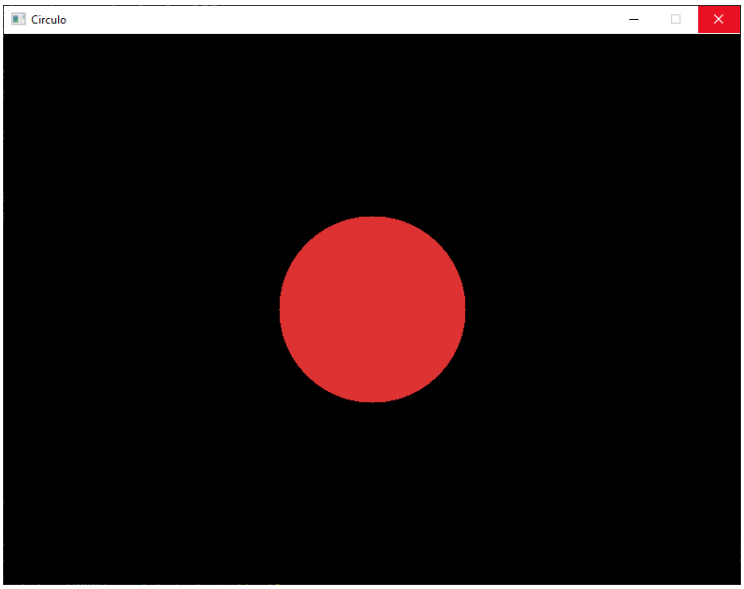
```
void desenharCirculo(SDL_Renderer *renderer, int cx, int cy, int raio){
        for (int dy = -raio; dy <= raio; dy++){
                int dx = (int)SDL_sqrt((double)(raio * raio - dy * dy));
                SDL_RenderDrawLine(
                        renderer,
                        cx - dx, cy + dy,
                        cx + dx, cy + dy
                );
        }
}
```

```
        SDL_SetRenderDrawColor(renderer, 0, 0, 0, 255);
        SDL_RenderClear(renderer);

        SDL_SetRenderDrawColor(renderer, 220, 50, 50, 255);
        desenharCirculo(renderer, 400, 300, 100);

        SDL_RenderPresent(renderer);
        SDL_Delay(5000);
```

Fonte: Autoria própria.

A função `SDL_sqrt` é a implementação de raiz quadrada fornecida pela própria SDL2, garantindo compatibilidade entre plataformas sem a necessidade de in-

Figura 7.6: Janela SDL mostrando o círculo.



Fonte: Autoria própria.

cluir `math.h` separadamente.

Desenhando a Célula do Connect 4

Com os três elementos anteriores — retângulo, círculo e a lógica de cores — é possível construir a célula característica do Connect 4: um quadrado azul com um círculo escuro no centro, representando uma posição vazia do tabuleiro, mostrado na Figura 7.7.

A técnica consiste em desenhar primeiro o retângulo azul preenchido e, em seguida, desenhar um círculo escuro sobre ele. Como a SDL2 renderiza os elementos na ordem em que são chamados, o círculo fica por cima do retângulo, criando a aparência de um buraco, como é indicado no Algoritmo 85:

O centro do círculo é calculado somando metade do tamanho da célula às coordenadas de origem: `centroX = origemX + tamanho / 2`. Isso garante que o círculo fique sempre centralizado dentro do retângulo, independentemente de onde a célula estiver posicionada. No Connect 4 completo, essa mesma lógica é aplicada para cada uma das 42 células da matriz, com a cor do círculo variando conforme o valor armazenado em `tabuleiro[linha][col]`.

7.5 Lógica do Jogo

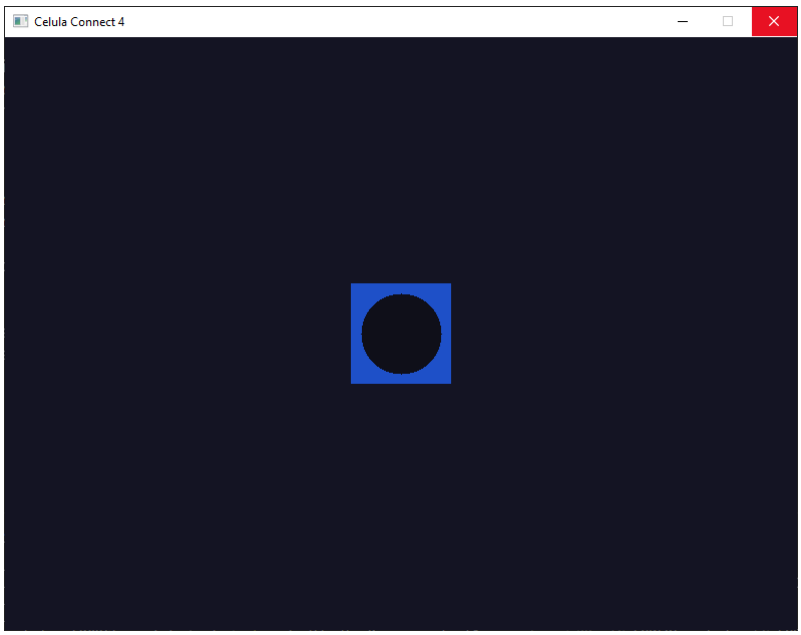
Esta seção apresenta a lógica central do Connect 4 de forma independente de qualquer biblioteca gráfica. Todo o código aqui pode ser reproduzido em qualquer linguagem de programação que suporte variáveis, matrizes, funções e estruturas de controle básicas.

Código-fonte 85 Algoritmo para desenhar a célula do Lig 4 dentro do SDL.

```
        int tamanho = 100;
        int origemX = 350;
        int origemY = 250;

        SDL_Rect celula = {origemX, origemY, tamanho, tamanho};
        SDL_SetRenderDrawColor(renderer, 30, 80, 200, 255);
        SDL_RenderFillRect(renderer, &celula);

        int centroX = origemX + tamanho / 2;
        int centroY = origemY + tamanho / 2;
        SDL_SetRenderDrawColor(renderer, 15, 15, 25, 255);
        desenharCirculo(renderer, centroX, centroY, 40);

        SDL_RenderPresent(renderer);
        SDL_Delay(6000);
```

Fonte: Autoria própria.

Figura 7.7: Janela SDL mostrando um quadrado do connect 4.



Fonte: Autoria própria.

A lógica do jogo pode ser dividida em quatro responsabilidades bem definidas: representar o estado do tabuleiro na memória, aplicar as regras de jogada, detectar o fim da partida e gerenciar a alternância de turnos.

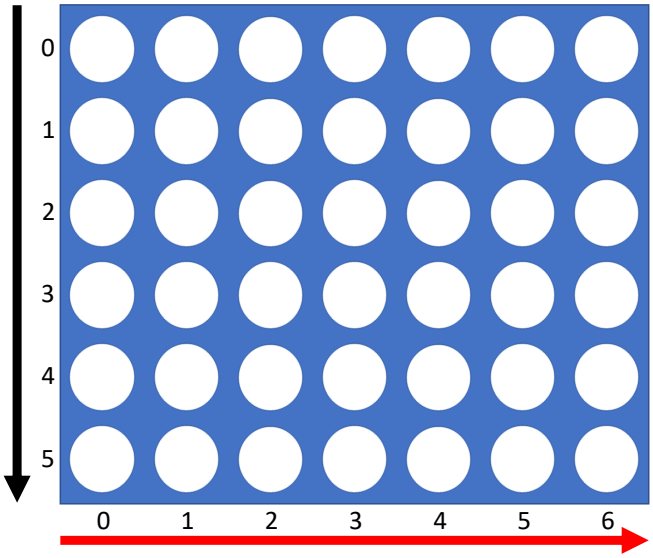
Representação do Tabuleiro

O tabuleiro do Connect 4 possui 6 linhas e 7 colunas, totalizando 42 posições [Hasbro, 2014]. Cada posição pode estar em um de três estados: vazia, ocupada pelo Jogador 1 ou ocupada pelo Jogador 2. A forma mais direta de representar isso na memória é uma matriz bidimensional de inteiros, onde cada valor indica o estado daquela célula.

Usamos três constantes para nomear esses estados, evitando o uso de números "mágicos" espalhados pelo código. Por exemplo, podemos definir `VAZIO = 0`, `JOGADOR1 = 1` e `JOGADOR2 = 2`.

A matriz é indexada como `tabuleiro[linha][coluna]`, onde a linha 0 representa o topo do tabuleiro e a linha 5 representa a base. Essa convenção é importante para a mecânica de gravidade, a Figura 7.8 representa a matriz do jogo.

Figura 7.8: Representação da matriz 6×7 com índices de linha e coluna.



Fonte: Autoria própria.

Para iniciar uma partida, todas as células devem ser preenchidas com `VAZIO`. Isso é feito percorrendo cada posição da matriz com dois laços aninhados — o externo itera sobre as linhas, e o interno sobre as colunas, atribuindo o valor `VAZIO` a cada posição.

Mecânica de Gravidade

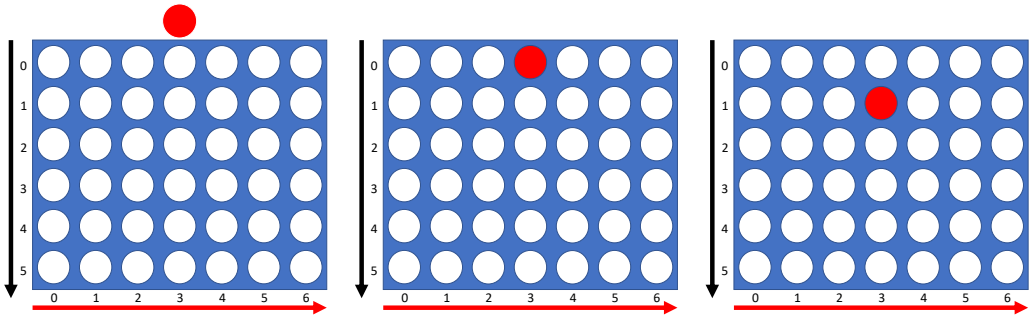
No Connect 4, o jogador escolhe apenas a coluna onde quer jogar. A peça então cai automaticamente até a posição mais baixa disponível naquela coluna, simulando a gravidade. Isso significa que nunca é possível colocar uma peça "flutuando" acima de uma célula vazia.

Para implementar isso, precisamos de duas funções. A primeira verifica se uma coluna ainda tem espaço disponível — basta checar se a célula do topo está vazia, pois as peças preenchem de baixo para cima.

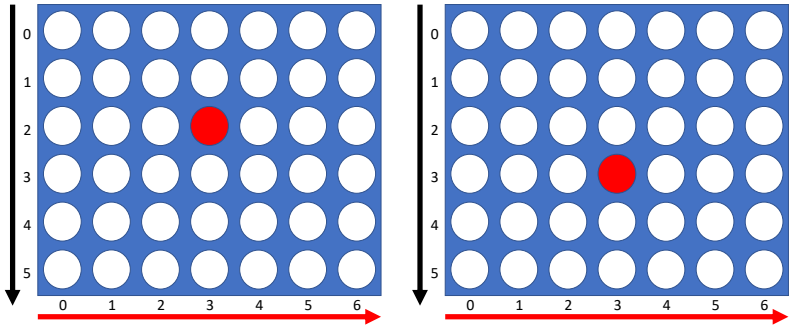
A segunda percorre a coluna de baixo para cima e retorna o índice da primeira linha vazia encontrada. Esse será o destino da próxima peça, como é indicado nas Figura 7.9.

Com essas duas funções, a colocação de uma peça se resume a validar a

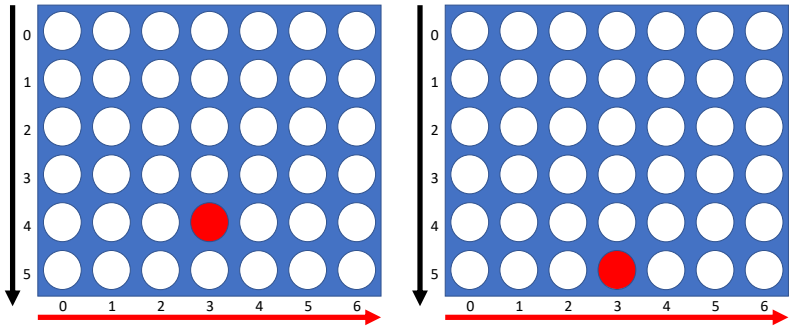
Figura 7.9: Sequência de queda de uma peça em uma coluna (mecânica de gravidade).



(a) 1° Passo. (b) 2° Passo. (c) 3° Passo.



(d) 4° Passo. (e) 5° Passo.



(f) 6° Passo. (g) Passo final.

Fonte: Autoria própria.

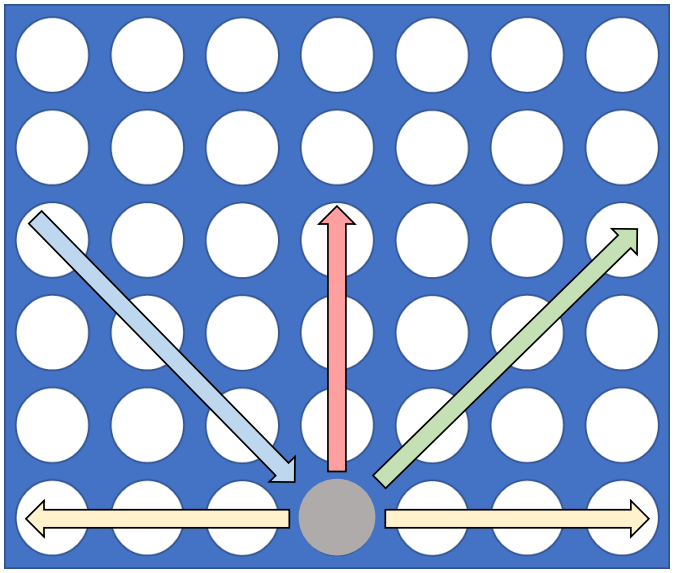
jogada e registrar o valor do jogador atual na posição encontrada. A função retorna `true` se a jogada foi bem-sucedida e `false` caso contrário, permitindo que quem a chama saiba se deve ou não trocar o turno.

Detecção de Vitória

Um jogador vence ao alinhar 4 peças consecutivas em qualquer uma das quatro direções possíveis: horizontal, vertical, diagonal descendente (\) e diagonal ascendente (/). A verificação percorre cada posição do tabuleiro como ponto de partida e, a

partir dela, compara os três próximos elementos naquela direção, uma representação disto seriam as Figuras 7.10 e 7.11.

Figura 7.10: As quatro direções possíveis para alinhar 4 peças consecutivas.



Fonte: Autoria própria.

A condição de vitória em cada direção pode ser expressa de forma geral: dados um ponto inicial `(linha, col)` e um passo `(dLinha, dCol)`, quatro células são iguais e não vazias se:

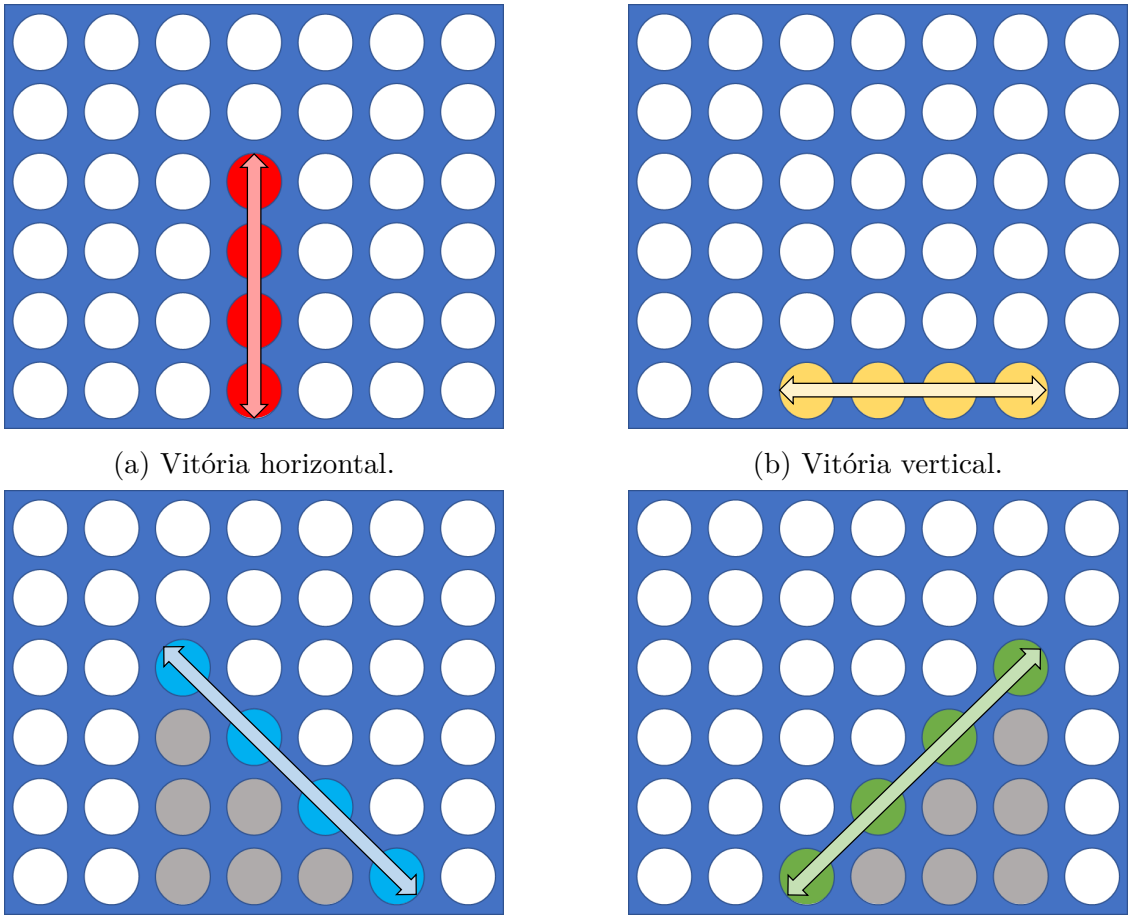
$$\texttt{tabuleiro[linha][col]} = \texttt{tabuleiro[linha + dLinha][col + dCol]} = \texttt{tabuleiro[linha + 2*dLinha][col + 2*dCol]} = \texttt{tabuleiro[linha + 3*dLinha][col + 3*dCol]} \neq \texttt{VAZIO}$$

Na prática, as quatro direções são verificadas em blocos separados. Os limites dos laços são ajustados em cada caso para evitar acesso fora dos limites da matriz. Por exemplo, na verificação horizontal, percorremos apenas até a coluna 3 (COLUNAS - 4), pois a partir da coluna 4 não é possível formar uma linha de 4 peças. A função retorna `1` ou `2` caso um dos jogadores tenha vencido, ou `0` caso nenhuma condição de vitória tenha sido encontrada.

Detecção de Empate

O empate ocorre quando todas as 42 células do tabuleiro estão preenchidas e nenhum jogador venceu. Como as peças sempre preenchem de baixo para cima, basta verificar se todas as células da linha 0 (o topo) estão ocupadas — se o topo de toda coluna tiver uma peça, não há mais jogadas possíveis, mostrado na Figura 7.12.

Figura 7.11: Exemplos de vitória nas quatro direções possíveis.



(a) Vitória horizontal.

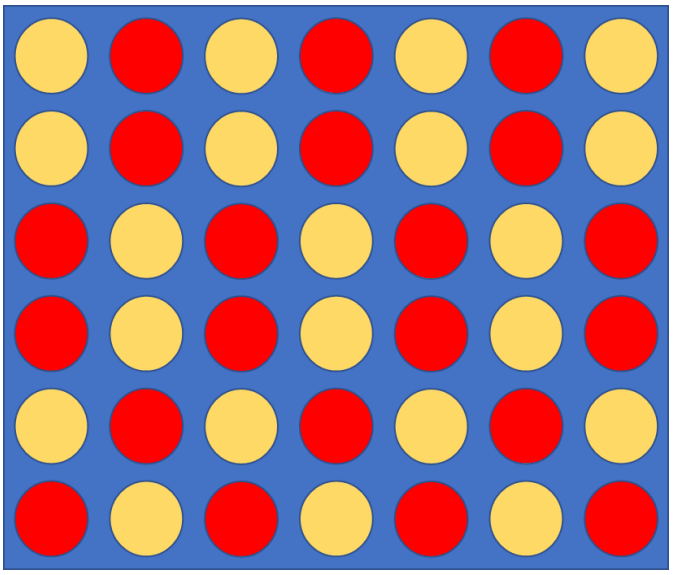
(b) Vitória vertical.

(c) Vitória diagonal \

(d) Vitória diagonal /.

Fonte: Autoria própria.

Figura 7.12: Exemplo de empate: tabuleiro completamente cheio sem vencedor.



Fonte: Autoria própria.

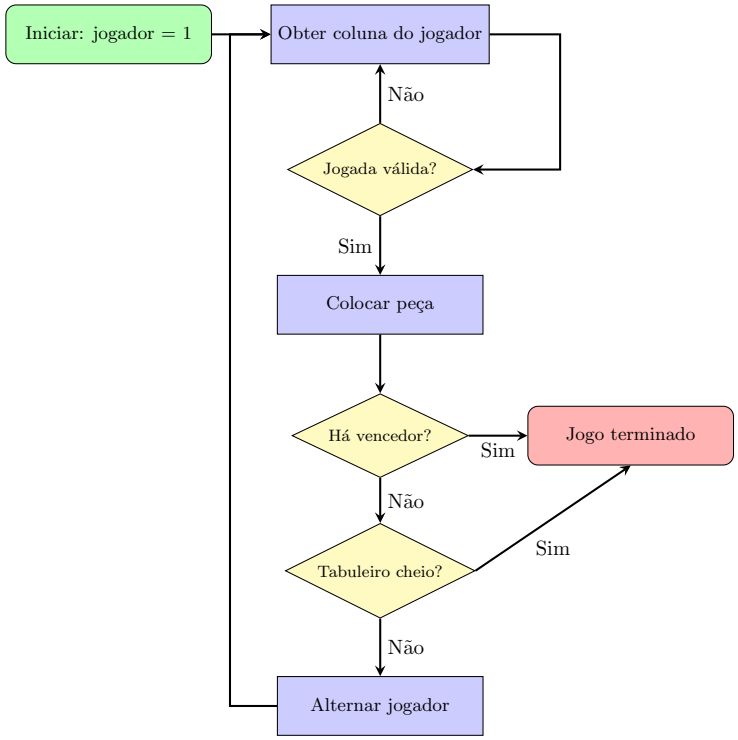
Alternância de Turnos

A cada jogada bem-sucedida, o turno passa para o outro jogador. Como os jogadores são representados pelos valores `1` e `2`, a alternância pode ser expressa de forma compacta: se o jogador atual é 1, o próximo é 2, e vice-versa.

Loop Principal da Lógica

Com todas as funções definidas, o fluxo completo de uma partida pode ser descrito pela seguinte Figura 7.13, que representa a sequência de decisões tomadas a cada rodada:

Figura 7.13: Diagrama de fluxo do algoritmo principal do Connect 4.

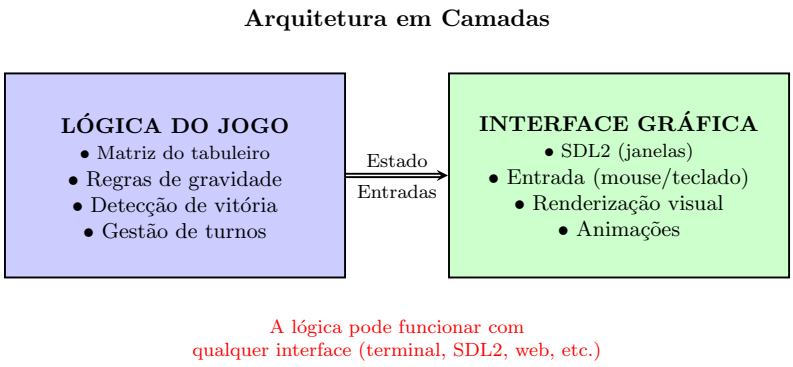


Fonte: Autoria própria.

1. Inicializa o jogador atual como Jogador 1
2. Enquanto o jogo não terminou:
    - Obtém a entrada do jogador (coluna escolhida)
    - Se a jogada for válida:
        - Coloca a peça no tabuleiro
        - Verifica se houve vencedor
        - Se houve vencedor, termina o jogo
        - Se não, verifica se o tabuleiro está cheio (empate)
        - Se não estiver cheio, alterna para o próximo jogador

A função de entrada representa qualquer mecanismo de entrada — pode ser leitura do terminal, um clique do mouse capturado pela SDL2, ou até uma decisão automática de uma inteligência artificial. O restante da lógica permanece idêntico em todos os casos, o que demonstra a vantagem de separar a lógica do jogo da camada de interface, mostrando na Figura 7.14.

Figura 7.14: Separação entre a lógica do jogo e a camada de interface gráfica.



Fonte: Autoria própria.

7.6 Conceitos Básicos de Programação em C++

Esta seção apresenta os fundamentos da linguagem C++ de forma progressiva, explicando cada conceito antes de ser utilizado no código do Connect 4. O objetivo é que mesmo pessoas sem experiência prévia em programação possam compreender como o jogo funciona.

O que é Programação?

Programação é a arte de dar instruções a um computador para que ele execute tarefas específicas. Pense na programação como escrever uma receita de bolo: você precisa listar os ingredientes (dados) e os passos (instruções) de forma clara e ordenada.

Em C++, essas instruções são escritas em um arquivo de texto com extensão `.cpp`. O computador não entende C++ diretamente — ele precisa de um *compilador*, que é um programa que traduz o código C++ para linguagem de máquina (zeros e uns) que o processador pode executar.

Estrutura Básica de um Programa C++

Todo programa em C++ tem uma estrutura fundamental. Vamos analisar o início do arquivo `connect4.cpp`, que seria o Algoritmo 86:

Bibliotecas (#include)

As primeiras linhas do código usam o comando `#include`. Pense nas bibliotecas como caixas de ferramentas prontas que você pode usar:

Código-fonte 86 Estrutura inicial de um programa C++ com bibliotecas.

```
#include <SDL2/SDL.h>
#include <SDL2/SDL_ttf.h>
#include <stdio.h>
#include <string.h>
```

Fonte: Autoria própria.

- `<SDL2/SDL.h>` — ferramentas para criar janelas e desenhar gráficos
- `<SDL2/SDL_ttf.h>` — ferramentas para escrever texto na tela
- `<stdio.h>` — ferramentas para imprimir mensagens no terminal
- `<string.h>` — ferramentas para manipular textos

Analogia: Se você fosse construir uma casa, não precisaria fabricar cada martelo e serrote. Você compraria uma caixa de ferramentas pronta. O `#include` é como abrir essa caixa de ferramentas.

Constantes (#define)

Constantes são nomes que damos a valores que não mudam durante o programa. No Connect 4, usamos constantes para definir o tamanho do tabuleiro com o Algoritmo 87:

Código-fonte 87 Definição de constantes no Connect 4.

```
#define LINHAS          6
#define COLUNAS         7
#define VAZIO           0
#define JOGADOR1        1
#define JOGADOR2        2
#define TAMANHO_CELULA 100
#define MARGEM_TOPO     120
#define RAIO_PECA       40
```

Fonte: Autoria própria.

Por que usar constantes? Imagine que você escreveu o número 6 em 50 lugares do código. Se decidir mudar o tabuleiro para 8 linhas, precisaria alterar 50 lugares! Com uma constante, você altera apenas uma vez:

- `#define LINHAS 6`
  Antes: tabuleiro de 6 linhas

- `#define LINHAS 8`
  Depois: tabuleiro de 8 linhas (mudou tudo automaticamente!)

Variáveis: Guardando Informações

Variáveis são como caixinhas onde guardamos informações. Cada variável tem três características:

1. **Tipo** — que tipo de informação guarda (número, texto, verdadeiro/falso)
2. **Nome** — como identificamos a caixinha
3. **Valor** — o que está guardado dentro

Tipos de Dados Primitivos

C++ tem tipos básicos de dados, chamados de *primitivos*:

`int` Guarda números inteiros (sem casas decimais): -5, 0, 1, 42, 1000

`bool` Guarda apenas dois valores: `true` (verdadeiro) ou `false` (falso)

`char` Guarda um único caractere: 'a', 'B', '9', '@'

`float` Guarda números decimais: 3.14, -0.5, 100.25

Exemplo prático:

Declarando e Usando Variáveis

No Connect 4, declaramos variáveis para guardar o estado do jogo, mostrado no Algoritmo 88:

Código-fonte 88 Variáveis globais do Connect 4.

```
1 int tabuleiro[LINHAS][COLUNAS];
2 int  jogadorAtual;
3 bool jogoTerminado;
4 int  vencedor;
5 char mensagem[100];
```

Fonte: Autoria própria.

Análise detalhada:

- `int tabuleiro[LINHAS][COLUNAS]` — uma matriz (tabela) com 6 linhas e 7 colunas
- `int jogadorAtual` — guarda quem está jogando (1 ou 2)
- `bool jogoTerminado` — indica se o jogo acabou
- `int vencedor` — guarda quem ganhou (0 = ninguém, 1 ou 2)
- `char mensagem[100]` — guarda um texto de até 100 caracteres

Matrizes: Tabelas de Dados

Uma matriz (ou *array bidimensional*) é uma tabela com linhas e colunas. É perfeita para representar o tabuleiro do Connect 4!

Como Funciona uma Matriz

- `int tabuleiro[6][7];`

Visualmente, o tabuleiro fica como a Tabela 7.15:

Figura 7.15: Representação do tabuleiro[6][7].

col0	**col1**	**col2**	**col3**	**col4**	**col5**	**col6**
0	0	0	0	0	0	0
0	0	0	0	0	0	0
0	0	0	0	0	0	0
0	0	0	0	0	0	0
0	0	0	0	0	0	0
0	0	0	0	0	0	0

Fonte: Autoria própria.

Acessando elementos da matriz:

- `tabuleiro[0][0] = 1;`
  Coloca 1 na linha 0, coluna 0

- `int valor = tabuleiro[5][2];`
  Lê o valor da linha 5, coluna 2

Importante: Em C++, a contagem começa do zero! A primeira linha é a linha 0, não a linha 1.

Estruturas de Controle: Tomando Decisões

Programas precisam tomar decisões e repetir tarefas. Para isso, usamos estruturas de controle.

Estrutura Condicional: if/else

O `if` executa um bloco de código **se** uma condição for verdadeira. O `else` executa **se** a condição for falsa. Um exemplo dele sendo usado dentro do código é apresentado no Algoritmo 89.

Sintaxe:

`if (condicao) {`
Código executado se a condição for verdadeira

`}else {`
Código executado se a condição for falsa
}

Exemplo prático:

1. `int idade = 18;`
2. `if (idade >= 18) {`
3. `printf("Você é maior de idade\n");`
4. `}else {`
5. `printf("Você é menor de idade\n");`
6. }

No Connect 4:

Código-fonte 89 Uso de if/else para validar jogadas.

```
bool colocarPeca(int col) {
    if (col < 0 || col >= COLUNAS) {
        return false;
    }

    if (!colunaDisponivel(col)) {
        return false;
    }

    int linha = obterLinhaLivre(col);
    tabuleiro[linha][col] = jogadorAtual;

    return true;
}
```

Fonte: Autoria própria.

Operadores de Comparação

Para criar condições, usamos operadores de comparação:

`==` Igual a (não confundir com `=` que é atribuição!)

`!=` Diferente de

`>` Maior que

`<` Menor que

`>=` Maior ou igual a

`<=` Menor ou igual a

Exemplo:

- if (idade == 18) { ... } // Se idade é exatamente 18
- if (idade != 18) { ... } // Se idade é diferente de 18
- if (idade > 18) { ... } // Se idade é maior que 18
- if (idade >= 18) { ... } // Se idade é 18 ou maior

Operadores Lógicos

Podemos combinar múltiplas condições:

`&&` E lógico (ambas condições devem ser verdadeiras)

`||` OU lógico (pelo menos uma condição deve ser verdadeira)

`!` NÃO lógico (inverte o valor)

Exemplo:

1. `if (idade >= 18 && temCarteira) {`
2. `printf("Pode dirigir\n");`
3. }
4. `if (idade < 18 || !temCarteira) {`
5. `printf("Não pode dirigir\n");`
6. }

Laços de Repetição: for e while

Muitas vezes precisamos repetir o mesmo código várias vezes. Para isso, usamos laços (*loops*).

Laço for

O `for` é usado quando sabemos quantas vezes queremos repetir. Um uso dele dentro do código é visto no Algoritmo 90.

Sintaxe:

- `for (inicializacao; condicao; incremento) {`
  Código
  }

Exemplo: contar de 0 a 5

1. `for (int i = 0; i < 6; i++) {`
2. `printf("%d\n", i);` // Imprime: 0, 1, 2, 3, 4, 5
3. }

Anatomia do for:

1. `int i = 0` — cria uma variável `i` e inicializa com 0
2. `i < 6` — condição: repete enquanto `i` for menor que 6

3. `i++` — incrementa `i` em 1 a cada volta (i = i + 1)

No Connect 4: percorrendo o tabuleiro

Código-fonte 90 Uso de laços for aninhados para percorrer matriz.

```
1 void inicializarTabuleiro() {
2     for (int linha = 0; linha < LINHAS; linha++) {
3         for (int col = 0; col < COLUNAS; col++) {
4             tabuleiro[linha][col] = VAZIO;
5         }
6     }
7 }
```

Fonte: Autoria própria.

Laços aninhados: Um `for` dentro de outro! O laço externo percorre as linhas, e o interno percorre as colunas. Isso visita cada célula da matriz.

Laço while

O `while` repete enquanto uma condição for verdadeira. Usamos quando não sabemos quantas vezes precisaremos repetir. Dentro do código, o Algoritmo 91 seria uma representação disso.

Sintaxe:

- `while (condicao) {`
  `// Código repetido enquanto condição for verdadeira`
  `}`

Exemplo: esperar até o usuário digitar 's'

1. `char resposta;`
2. `while (resposta != 's') {`
3. `  printf("Digite 's' para continuar: ");`
4. `  scanf("%c", &resposta);`
5. `}`

No Connect 4: loop principal do jogo

Funções: Organizando o Código

Funções são blocos de código com um nome que executam uma tarefa específica. São como "mini-programas"dentro do programa principal.

Código-fonte 91 Loop principal do jogo usando while.

```
    while (rodando) {
        while (SDL_PollEvent(&evento)) {
            if (evento.type == SDL_QUIT) {
                rodando = false;
            }
            if (evento.type == SDL_MOUSEMOTION) {
                mouseX = evento.motion.x;
            }
            if (evento.type == SDL_MOUSEBUTTONDOWN) {
                if (evento.button.button == SDL_BUTTON_LEFT &&
↪   !jogoTerminado) {
                    int col = evento.button.x / TAMANHO_CELULA;

                    if (colocarPeca(col)) {
                        vencedor = verificarVencedor();

                        if (vencedor != 0) {
                            jogoTerminado = true;
                        } else if (tabuleiroCheio()) {
                            jogoTerminado = true;
                            vencedor = 0;
                        } else {
                            trocarTurno();
                        }
                    }
                }
            }
            if (evento.type == SDL_KEYDOWN) {
                if (evento.key.keysym.sym == SDLK_r) {
                    reiniciarJogo();
                }
                if (evento.key.keysym.sym == SDLK_ESCAPE) {
                    rodando = false;
                }
            }
        }
        desenharTabuleiro(renderer, mouseX);

        if (fonteGrande && fontePequena) {
            desenharInterface(renderer, fonteGrande, fontePequena);
        }
        SDL_RenderPresent(renderer);
        SDL_Delay(16);
    }
```

Fonte: Autoria própria.

Por que usar funções?

- **Organização** — dividem o código em partes menores e mais fáceis de entender
- **Reutilização** — escrevem uma vez, usam várias vezes
- **Manutenção** — facilitam encontrar e corrigir erros

Estrutura de uma Função

1. `tipo_retorno nomeDaFuncao(parametros)` {
2. Código que a função executa
3. `return valor`; (Opcional: depende do tipo_retorno)
4. }

Componentes:

tipo_retorno O tipo de valor que a função retorna (`void`, `int`, `bool`, etc.)

nomeDaFuncao Nome que usamos para chamar a função

parâmetros Valores que a função recebe (opcional)

return Valor que a função devolve (opcional para `void`)

Tipos de Retorno

`void` A função não retorna nada (apenas executa tarefas)

`int` A função retorna um número inteiro

`bool` A função retorna `true` ou `false`

Exemplos:

O seguinte Algoritmo 92 tem presente alguns exemplos de funções:

Funções do Connect 4

Vamos analisar as principais funções do jogo nos Algoritmos 93, 94, 95 e 96:

1. Função que inicializa o tabuleiro:

Análise:

- `void` — não retorna nada

Código-fonte 92 Exemplos de funções.

```
// Função que não retorna nada
void imprimirMensagem() {
    printf("Olá, mundo!\n");
}
// Função que retorna um número
int somar(int a, int b) {
    return a + b;
}
// Função que retorna verdadeiro/falso
bool ehPar(int numero) {
    return (numero % 2 == 0);
}
```

Fonte: Autoria própria.

Código-fonte 93 Função inicializarTabuleiro.

```
void inicializarTabuleiro() {
    for (int linha = 0; linha < LINHAS; linha++) {
        for (int col = 0; col < COLUNAS; col++) {
            tabuleiro[linha][col] = VAZIO;
        }
    }
}
```

Fonte: Autoria própria.

- `()` — não recebe parâmetros
- Usa dois `for` para colocar `VAZIO` em cada célula

2. Função que verifica se uma coluna está disponível:

Código-fonte 94 Função colunaDisponivel.

```
bool colunaDisponivel(int col) {
    return tabuleiro[0][col] == VAZIO;
}
```

Fonte: Autoria própria.

Análise:

- `bool` — retorna `true` ou `false`
- `int col` — recebe o número da coluna como parâmetro
- Verifica se a linha 0 (topo) está vazia

3. Função que coloca uma peça:

Análise:

Código-fonte 95 Função colocarPeca.

```
bool colocarPeca(int col) {
    if (col < 0 || col >= COLUNAS) {
        return false;
    }

    if (!colunaDisponivel(col)) {
        return false;
    }

    int linha = obterLinhaLivre(col);
    tabuleiro[linha][col] = jogadorAtual;

    return true;
}
```

Fonte: Autoria própria.

- `bool` — retorna `true` se conseguiu, `false` se falhou
- `int col` — recebe a coluna onde colocar a peça
- Usa `if/else` para validar a jogada
- Retorna `false` se a coluna for inválida ou cheia

Chamando Funções

Para usar uma função, precisamos *chamá-la* pelo nome:

Código-fonte 96 Chamando uma função dentro do código.

```
int somar(int a, int b) {
    return a + b;
}
```

```
    int resultado = somar(5, 3);  // resultado = 8
    printf("A soma é: %d\n", resultado);
```

Fonte: Autoria própria.

Operador Ternário: if/else em uma linha

C++ tem uma forma compacta de escrever `if/else` em uma única linha:

Sintaxe:

`variavel` = (`condicao`) ? `valor_se_verdadeiro` : `valor_se_falso`;

Exemplo:

1. `int idade` = 20;

2. `string status = (idade >= 18) ? "maior" : "menor";`
   Se idade ≥ 18, status = "maior"
   Se idade < 18, status = "menor"

No Connect 4: trocando o turno

O seguinte Algoritmo 97 tem a representação de troca de turno.

Código-fonte 97 Uso do operador ternário para alternar jogadores.

```
1 void trocarTurno() {
2     jogadorAtual = (jogadorAtual == JOGADOR1) ? JOGADOR2 : JOGADOR1;
3 }
```

Fonte: Autoria própria.

Equivalente com if/else:

1. `if (jogadorAtual == JOGADOR1) {`
2. `    jogadorAtual = JOGADOR2;`
3. `} else {`
4. `    jogadorAtual = JOGADOR1;`
5. `}`

A Função Principal: main()

Todo programa em C++ precisa de uma função chamada `main()`. É o ponto de partida — quando você executa o programa, o computador vai direto para o `main()`.

Estrutura típica do main():

1. Inicializar bibliotecas e recursos
2. Configurar o estado inicial
3. Executar o loop principal
4. Liberar recursos e encerrar

No Connect 4:

Tabela 7.1: Resumo dos conceitos de programação C++.

Conceito	Descrição
`#include`	Importa bibliotecas de código pronto
`#define`	Cria constantes (valores fixos)
`int`	Variável para números inteiros
`bool`	Variável para verdadeiro/falso
`char[]`	Variável para texto (array de caracteres)
`int[][]`	Matriz (tabela bidimensional)
`if/else`	Estrutura condicional (tomar decisões)
`for`	Laço de repetição com contador
`while`	Laço de repetição com condição
`funcao()`	Bloco de código reutilizável
`return`	Retorna um valor de uma função
`? :`	Operador ternário (if/else compacto)

Resumo dos Conceitos Aprendidos

A seguinte Tabela 7.1 tem um resumem do que foi apresentado nesta seção:

Lembre-se: Programação se aprende programando! Não tenha medo de experimentar, cometer erros e aprender com eles. Cada erro é uma oportunidade de aprendizado.

7.7 Código Completo do Connect 4

Esta seção apresenta o código completo do jogo Connect 4 desenvolvido no minicurso, dividido em blocos que correspondem à ordem de apresentação em aula. O arquivo completo (`connect4.cpp`) pode ser compilado com os comandos mostrados nos Algoritmos 98 e 99:

Código-fonte 98 Compilação do connect4 no Windows.

```
$ g++ connect4.cpp -o connect4.exe -lmingw32 -IC:\SDL2\include ^
↪  -LC:\SDL2\lib -lSDL2main -lSDL2 -lSDL2_ttf -mwindows
```

Fonte: Autoria própria.

Código-fonte 99 Compilação do connect4 no Linux.

```
$ # Compilação do Connect 4 no Linux usando pkg-config
$ g++ connect4.cpp -o connect4 $(pkg-config --cflags --libs SDL2
↪  SDL2_ttf)

$ # Alternativa: compilação explícita (ajuste os caminhos conforme sua
↪  instalação)
$ # g++ connect4.cpp -o connect4 -I/usr/include/SDL2 -L/usr/lib -lSDL2
↪  -lSDL2_ttf
```

Fonte: Autoria própria.

Includes e Constantes

Os `#include` importam as bibliotecas necessárias, e as constantes com `#define` nomeiam os valores fixos do jogo, facilitando a leitura e manutenção do código, como mostrado no Algoritmo 100.

Código-fonte 100 Includes e Constantes.

```
1  #include <SDL2/SDL_ttf.h>
2  #include <stdio.h>
3  #include <string.h>
4  #define COLUNAS        7    // O tabuleiro tem 7 colunas
5  #define VAZIO          0    // Uma celula vazia vale 0
6  #define JOGADOR1       1    // O jogador 1 vale 1
7  #define JOGADOR2       2    // O jogador 2 vale 2
8
```

Fonte: Autoria própria.

Variáveis Globais e Matriz

As variáveis globais guardam o estado atual da partida. O tabuleiro é declarado como uma matriz bidimensional de inteiros, onde cada posição representa uma célula, mostrado no Algoritmo 101.

Código-fonte 101 Matriz e variáveis globais.

```
105 int tabuleiro[LINHAS][COLUNAS];
106
107 // Variaveis que guardam o estado atual do jogo
108 int  jogadorAtual;   // Quem esta jogando agora: 1 ou 2
109 bool jogoTerminado;  // O jogo acabou? true ou false
110 int  vencedor;       // Quem ganhou: 0 = ninguem, 1 ou 2 = vencedor
111 char mensagem[100];  // Texto para mostrar na tela
```

Fonte: Autoria própria.

Funções de Lógica do Jogo

Cada função tem uma responsabilidade única. Juntas, elas implementam todas as regras do Connect 4: inicializar o tabuleiro, colocar peças, simular a gravidade, trocar turnos e detectar vitória ou empate, essas funções podem ser vistas nos Algoritmos 102, 103, 104 e 105.

Código-fonte 102 Lógicas do Jogo.

```
void inicializarTabuleiro() {
    for (int linha = 0; linha < LINHAS; linha++) {
        for (int col = 0; col < COLUNAS; col++) {
            tabuleiro[linha][col] = VAZIO;
        }
    }
}
bool colunaDisponivel(int col) {
    return tabuleiro[0][col] == VAZIO;
}
int obterLinhaLivre(int col) {
    for (int linha = LINHAS - 1; linha >= 0; linha--) {
        if (tabuleiro[linha][col] == VAZIO) {
            return linha;  // Achou! Retorna o numero da linha
        }
    }
    return -1;  // Nao achou (coluna cheia) - nao deve acontecer
}
bool colocarPeca(int col) {
    // Verifica se a coluna existe no tabuleiro
    if (col < 0 || col >= COLUNAS) {
        return false;  // Coluna invalida, nao faz nada
    }

    // Verifica se ha espaco na coluna
    if (!colunaDisponivel(col)) {
        return false;  // Coluna cheia, nao faz nada
    }

    // Encontra a linha mais baixa disponivel
    int linha = obterLinhaLivre(col);

    // Coloca a peca do jogador atual nessa posicao
    tabuleiro[linha][col] = jogadorAtual;

    return true;  // Deu certo!
}
```

Fonte: Autoria própria.

Código-fonte 103 Função verificarVendedor() - parte 1.

```
int verificarVencedor() {
    // Verifica horizontal: 4 pecas na mesma linha
    for (int linha = 0; linha < LINHAS; linha++) {
        for (int col = 0; col <= COLUNAS - 4; col++) {
            int peca = tabuleiro[linha][col];
            if (peca != VAZIO &&
                peca == tabuleiro[linha][col+1] &&
                peca == tabuleiro[linha][col+2] &&
                peca == tabuleiro[linha][col+3]) {
                return peca;
            }
        }
    }

    // Verifica vertical: 4 pecas na mesma coluna
    for (int linha = 0; linha <= LINHAS - 4; linha++) {
        for (int col = 0; col < COLUNAS; col++) {
            int peca = tabuleiro[linha][col];
            if (peca != VAZIO &&
                peca == tabuleiro[linha+1][col] &&
                peca == tabuleiro[linha+2][col] &&
                peca == tabuleiro[linha+3][col]) {
                return peca;
            }
        }
    }

    // Verifica diagonal para baixo-direita (\)
    for (int linha = 0; linha <= LINHAS - 4; linha++) {
        for (int col = 0; col <= COLUNAS - 4; col++) {
            int peca = tabuleiro[linha][col];
            if (peca != VAZIO &&
                peca == tabuleiro[linha+1][col+1] &&
                peca == tabuleiro[linha+2][col+2] &&
                peca == tabuleiro[linha+3][col+3]) {
                return peca;
            }
        }
    }
```

Fonte: Autoria própria.

Código-fonte 104 Função verificarVendedor() - parte 2.

```
40      // Verifica diagonal para baixo-esquerda (/)
41      for (int linha = 0; linha <= LINHAS - 4; linha++) {
42          for (int col = 3; col < COLUNAS; col++) {
43              int peca = tabuleiro[linha][col];
44              if (peca != VAZIO &&
45                  peca == tabuleiro[linha+1][col-1] &&
46                  peca == tabuleiro[linha+2][col-2] &&
47                  peca == tabuleiro[linha+3][col-3]) {
48                  return peca;
49              }
50          }
51      }
52
53      return 0;  // Nenhum vencedor encontrado
54  }
```

Fonte: Autoria própria.

Código-fonte 105 Lógicas do Jogo.

```
1   bool tabuleiroCheio() {
2       for (int col = 0; col < COLUNAS; col++) {
3           if (colunaDisponivel(col)) {
4               return false;  // Encontrou espaco, nao esta cheio
5           }
6       }
7       return true;  // Nenhum espaco encontrado = cheio
8   }
9   void trocarTurno() {
10      jogadorAtual = (jogadorAtual == JOGADOR1) ? JOGADOR2 : JOGADOR1;
11  }
12  void reiniciarJogo() {
13      inicializarTabuleiro();   // Limpa o tabuleiro
14      jogadorAtual  = JOGADOR1; // Jogador 1 comeca sempre
15      jogoTerminado = false;    // O jogo nao acabou ainda
16      vencedor      = 0;        // Nenhum vencedor ainda
17      strcpy(mensagem, "Vez do Jogador 1 (Vermelho)");
18  }
```

Fonte: Autoria própria.

Interface Gráfica com SDL2

As funções gráficas usam a SDL2 para desenhar o tabuleiro (Algoritmos 107, 108 e 109), as peças e os textos (Algoritmos 110 e 111) na janela. Como a SDL2 não possui função nativa para círculos, implementamos `desenharCirculo()` com o Teorema de Pitágoras no Algoritmo 106.

Código-fonte 106 Função para desenhar Círculos.

```
1 void desenharCirculo(SDL_Renderer* renderer, int cx, int cy, int raio) {
2     for (int dy = -raio; dy <= raio; dy++) {
3         // dx = largura da linha nessa altura (Pitagoras: dx^2 + dy^2 =
  ↪  raio^2)
4         int dx = (int)SDL_sqrt((double)(raio * raio - dy * dy));
5         SDL_RenderDrawLine(renderer, cx - dx, cy + dy, cx + dx, cy + dy);
6     }
7 }
```

Fonte: Autoria própria.

Função Principal `main()`

O `main()` é o ponto de entrada do programa. Inicializa a SDL2, cria a janela e o renderer, inicia o jogo e roda o loop principal de eventos até o jogador fechar a aplicação, mostrado nos Algoritmos 112, 113, 114 e 115.

7.8 Conclusão

Este minicurso apresentou uma abordagem prática e acessível para ensinar programação em C++ através do desenvolvimento do jogo Connect 4, utilizando a biblioteca SDL2 para criar uma interface gráfica interativa. Os conceitos fundamentais de programação foram introduzidos de forma progressiva, desde variáveis e estruturas de controle até funções e lógica de jogo, sempre com aplicação imediata no projeto prático.

O trabalho realizado vai além da simples recreação do jogo Connect 4. Enquanto o minicurso foca nos conceitos básicos de programação e na implementação da lógica do jogo para dois jogadores humanos, o trabalho referenciado na bibliografia Soares [2020], desenvolvido por Pedro Michael Santos Soares, inclui um componente significativo que não foi abordado neste material: um agente de inteligência artificial capaz de jogar contra o jogador humano.

Este agente de IA, descrito na metodologia de Soares, implementa algoritmos de busca e avaliação de posições que permitem ao computador tomar decisões estratégicas durante o jogo, proporcionando um desafio adicional e mais avançado para

Código-fonte 107 Função para desenhar o tabuleiro - parte 1.

```
void desenharTabuleiro(SDL_Renderer* renderer, int mouseX) {

    // Descobre em qual coluna o mouse esta
    int colMouse = mouseX / TAMANHO_CELULA;
    if (colMouse >= COLUNAS) colMouse = COLUNAS - 1;
    if (colMouse < 0)        colMouse = 0;

    // Pinta o fundo da janela de preto/escuro
    SDL_SetRenderDrawColor(renderer, 20, 20, 35, 255);
    SDL_RenderClear(renderer);

    // Mostra uma peca "fantasma" seguindo o mouse (preview da jogada)
    if (!jogoTerminado && colunaDisponivel(colMouse)) {
        if (jogadorAtual == JOGADOR1)
            SDL_SetRenderDrawColor(renderer, 220, 50, 50, 255);   //
↪   Vermelho
        else
            SDL_SetRenderDrawColor(renderer, 240, 200, 50, 255);  //
↪   Amarelo

        int preX = colMouse * TAMANHO_CELULA + TAMANHO_CELULA / 2;
        int preY = MARGEM_TOPO / 2;
        desenharCirculo(renderer, preX, preY, RAIO_PECA);
    }

    // Desenha o fundo azul do tabuleiro
    SDL_Rect retTabuleiro = {
        0,
        MARGEM_TOPO,
        COLUNAS * TAMANHO_CELULA,
        LINHAS  * TAMANHO_CELULA
    };
    SDL_SetRenderDrawColor(renderer, 30, 80, 200, 255);
    SDL_RenderFillRect(renderer, &retTabuleiro);
```

Fonte: Autoria própria.

Código-fonte 108 Função para desenhar o tabuleiro - parte 2.

```
33      // Percorre a matriz e desenha cada celula
34      for (int linha = 0; linha < LINHAS; linha++) {
35          for (int col = 0; col < COLUNAS; col++) {
36
37              // Calcula o centro em pixels desta celula
38              int cx = col   * TAMANHO_CELULA + TAMANHO_CELULA / 2;
39              int cy = MARGEM_TOPO + linha * TAMANHO_CELULA +
    ↪  TAMANHO_CELULA / 2;
40
41              if (tabuleiro[linha][col] == VAZIO) {
42                  // Celula vazia: buraco escuro
43                  SDL_SetRenderDrawColor(renderer, 15, 15, 25, 255);
44                  desenharCirculo(renderer, cx, cy, RAIO_PECA);
45
46              } else if (tabuleiro[linha][col] == JOGADOR1) {
47                  // Peca vermelha (Jogador 1)
48                  SDL_SetRenderDrawColor(renderer, 220, 50, 50, 255);
49                  desenharCirculo(renderer, cx, cy, RAIO_PECA);
50                  // Brilho para dar efeito de profundidade
51                  SDL_SetRenderDrawColor(renderer, 255, 120, 120, 255);
52                  desenharCirculo(renderer, cx - 8, cy - 8, RAIO_PECA / 3);
53
54              } else {
55                  // Peca amarela (Jogador 2)
56                  SDL_SetRenderDrawColor(renderer, 240, 200, 50, 255);
57                  desenharCirculo(renderer, cx, cy, RAIO_PECA);
58                  // Brilho para dar efeito de profundidade
59                  SDL_SetRenderDrawColor(renderer, 255, 240, 150, 255);
60                  desenharCirculo(renderer, cx - 8, cy - 8, RAIO_PECA / 3);
61              }
62          }
63      }
```

Fonte: Autoria própria.

quem já dominou os conceitos básicos apresentados aqui. A inclusão de tal agente representaria uma evolução natural deste minicurso para um nível intermediário ou avançado, onde os estudantes poderiam aprender sobre algoritmos de decisão, funções de heurística e otimização de desempenho.

Ao separar claramente a lógica do jogo da camada de interface gráfica (como demonstrado na Figura 7.14), este trabalho estabelece uma base sólida para futuras extensões, incluindo precisamente a implementação de agentes de IA que possam ser facilmente integrados à arquitetura existente sem modificar a lógica central do jogo.

Assim, enquanto este minicurso fornece as ferramentas essenciais para entender e implementar o Connect 4 em C++, o trabalho de Soares [2020] mostra um

Código-fonte 109 Função para desenhar o tabuleiro - parte 3.

```
64      // Desenha as linhas da grade por cima do tabuleiro
65      SDL_SetRenderDrawColor(renderer, 20, 60, 170, 255);
66      for (int col = 0; col <= COLUNAS; col++) {
67          int x = col * TAMANHO_CELULA;
68          SDL_RenderDrawLine(renderer, x, MARGEM_TOPO, x, MARGEM_TOPO +
    ↪   LINHAS * TAMANHO_CELULA);
69      }
70      for (int linha = 0; linha <= LINHAS; linha++) {
71          int y = MARGEM_TOPO + linha * TAMANHO_CELULA;
72          SDL_RenderDrawLine(renderer, 0, y, COLUNAS * TAMANHO_CELULA, y);
73      }
74  }
```

Fonte: Autoria própria.

Código-fonte 110 Função para desenhar o texto.

```
1   void desenharTexto(SDL_Renderer* renderer, TTF_Font* fonte,
2                      const char* texto, int x, int y,
3                      int r, int g, int b) {
4       SDL_Color cor       = {(Uint8)r, (Uint8)g, (Uint8)b, 255};
5       SDL_Surface* surf   = TTF_RenderText_Blended(fonte, texto, cor);
6       if (!surf) return;
7       SDL_Texture* tex    = SDL_CreateTextureFromSurface(renderer, surf);
8       if (tex) {
9           SDL_Rect dest   = {x, y, surf->w, surf->h};
10          SDL_RenderCopy(renderer, tex, NULL, &dest);
11          SDL_DestroyTexture(tex);
12      }
13      SDL_FreeSurface(surf);
14  }
```

Fonte: Autoria própria.

caminho para aprofundar esses conhecimentos através da adição de componentes inteligentes que transformam um simples jogo em um oponente desafiador capaz de aprender e se adaptar às estratégias do jogador humano.

Código-fonte 111 Função para desenhar a interface.

```
void desenharInterface(SDL_Renderer* renderer,
                       TTF_Font* fonteGrande,
                       TTF_Font* fontePequena) {

    // Titulo do jogo
    desenharTexto(renderer, fonteGrande, "QUATRO EM LINHA", 10, 5, 200,
↪   220, 255);

    if (jogoTerminado) {
        // Mensagem de vitoria ou empate
        if (vencedor == JOGADOR1)
            desenharTexto(renderer, fontePequena, "JOGADOR 1 VENCEU!",
↪   10, 65, 220, 80, 80);
        else if (vencedor == JOGADOR2)
            desenharTexto(renderer, fontePequena, "JOGADOR 2 VENCEU!",
↪   10, 65, 240, 200, 50);
        else
            desenharTexto(renderer, fontePequena, "EMPATE!", 10, 65, 180,
↪   180, 180);

        desenharTexto(renderer, fontePequena,
                      "Pressione R para reiniciar", 310, 65, 150, 200,
↪   255);
    } else {
        // Mostra de quem e a vez
        if (jogadorAtual == JOGADOR1)
            desenharTexto(renderer, fontePequena,
                          "Vez do: JOGADOR 1 (Vermelho)", 10, 65, 220,
↪   100, 100);
        else
            desenharTexto(renderer, fontePequena,
                          "Vez do: JOGADOR 2 (Amarelo)", 10, 65, 240,
↪   200, 50);

        desenharTexto(renderer, fontePequena,
                      "Clique = jogar  |  R = reiniciar  |  ESC = sair",
                      240, 65, 140, 160, 200);
    }
}
```

Fonte: Autoria própria.

Código-fonte 112 Função `main()` - parte 1.

```
int main(int argc, char* argv[]) {

    // --- PASSO 1: Inicializar a SDL2 ---
    // SDL_Init "liga" a biblioteca. SDL_INIT_VIDEO ativa o modo grafico.
    if (SDL_Init(SDL_INIT_VIDEO) < 0) {
        printf("Erro ao inicializar SDL2: %s\n", SDL_GetError());
        return 1;  // Retornar 1 no main() significa que algo deu errado
    }

    // Inicializa o modulo de texto (SDL_ttf)
    if (TTF_Init() < 0) {
        printf("Erro ao inicializar SDL_ttf: %s\n", TTF_GetError());
        SDL_Quit();
        return 1;
    }

    // --- PASSO 2: Calcular o tamanho da janela ---
    int largura = COLUNAS * TAMANHO_CELULA;          // 7 * 100 = 700
↪   pixels
    int altura  = LINHAS  * TAMANHO_CELULA + MARGEM_TOPO; // 6 * 100 +
↪   120 = 720 pixels

    // --- PASSO 3: Criar a janela ---
    SDL_Window* janela = SDL_CreateWindow(
        "Quatro em Linha - Minicurso C++", // Titulo da janela
        SDL_WINDOWPOS_CENTERED,            // Centraliza na tela (eixo X)
        SDL_WINDOWPOS_CENTERED,            // Centraliza na tela (eixo Y)
        largura,                           // Largura em pixels
        altura,                            // Altura em pixels
        SDL_WINDOW_SHOWN                   // Mostra a janela
↪   imediatamente
    );

    if (!janela) {
        printf("Erro ao criar janela: %s\n", SDL_GetError());
        TTF_Quit();
        SDL_Quit();
        return 1;
    }
```

Fonte: Autoria própria.

Código-fonte 113 Função `main()` - parte 2.

```
37      // --- PASSO 4: Criar o Renderer ---
38      // O renderer e o "pincel" que usamos para desenhar dentro da janela
39      SDL_Renderer* renderer = SDL_CreateRenderer(janela, -1,
    ↪   SDL_RENDERER_ACCELERATED);
40
41      if (!renderer) {
42          printf("Erro ao criar renderer: %s\n", SDL_GetError());
43          SDL_DestroyWindow(janela);
44          TTF_Quit();
45          SDL_Quit();
46          return 1;
47      }
48
49      // --- PASSO 5: Carregar as fontes de texto ---
50      // Tentamos carregar fontes do Windows primeiro, depois Linux e macOS
51      TTF_Font* fonteGrande  =
    ↪   TTF_OpenFont("C:/Windows/Fonts/arialbd.ttf", 38);
52      TTF_Font* fontePequena = TTF_OpenFont("C:/Windows/Fonts/arial.ttf",
    ↪    22);
53
54      if (!fonteGrande)
55          fonteGrande  = TTF_OpenFont("/usr/share/fonts/truetype/liberatio
    ↪   n/LiberationSans-Bold.ttf",
    ↪   38);
56      if (!fontePequena)
57          fontePequena = TTF_OpenFont("/usr/share/fonts/truetype/liberatio
    ↪   n/LiberationSans-Regular.ttf",
    ↪   22);
58
59      if (!fonteGrande)
60          fonteGrande  =
    ↪   TTF_OpenFont("/System/Library/Fonts/Helvetica.ttc", 38);
61      if (!fontePequena)
62          fontePequena =
    ↪   TTF_OpenFont("/System/Library/Fonts/Helvetica.ttc", 22);
63
64      if (!fonteGrande || !fontePequena)
65          printf("Aviso: fonte nao carregada. Texto nao sera exibido.\n");
66
67      // --- PASSO 6: Iniciar o jogo ---
68      reiniciarJogo();
69
70      // Guarda a posicao X do mouse
71      int mouseX = 0;
```

Fonte: Autoria própria.

Código-fonte 114 Função `main()` - parte 3.

```
72      bool rodando = true;
73      SDL_Event evento;  // Variavel que guarda o evento capturado
74      while (rodando) {
75          // --- PARTE 1: PROCESSAR EVENTOS ---
76          // SDL_PollEvent captura o proximo evento da fila de eventos
77          while (SDL_PollEvent(&evento)) {
78
79              // Evento: jogador clicou no X para fechar a janela
80              if (evento.type == SDL_QUIT) {
81                  rodando = false;
82              }
83              // Evento: jogador moveu o mouse (atualizamos a posicao X)
84              if (evento.type == SDL_MOUSEMOTION) {
85                  mouseX = evento.motion.x;
86              }
87              // Evento: jogador clicou com o botao esquerdo do mouse
88              if (evento.type == SDL_MOUSEBUTTONDOWN) {
89                  if (evento.button.button == SDL_BUTTON_LEFT &&
    ↪  !jogoTerminado) {
90
91                      // Descobre em qual coluna foi o clique
92                      // (dividimos a posicao X pelo tamanho da celula)
93                      int col = evento.button.x / TAMANHO_CELULA;
94
95                      // Tenta colocar a peca nessa coluna
96                      if (colocarPeca(col)) {
97
98                          // Verifica se alguem ganhou apos essa jogada
99                          vencedor = verificarVencedor();
100
101                         if (vencedor != 0) {
102                             jogoTerminado = true;      // Alguem ganhou!
103                         } else if (tabuleiroCheio()) {
104                             jogoTerminado = true;      // Empate!
105                             vencedor = 0;
106                         } else {
107                             trocarTurno();             // Jogo continua,
    ↪  troca vez
108                         }
109                     }
110                 }
111             }
```

Fonte: Autoria própria.

Código-fonte 115 Função `main()` - parte 4.

```
112             // Evento: jogador pressionou uma tecla do teclado
113             if (evento.type == SDL_KEYDOWN) {
114                 if (evento.key.keysym.sym == SDLK_r) {
115                     reiniciarJogo();   // Tecla R: reinicia o jogo
116                 }
117                 if (evento.key.keysym.sym == SDLK_ESCAPE) {
118                     rodando = false;   // Tecla ESC: fecha o jogo
119                 }
120             }
121         }
122 
123         // --- PARTE 2: DESENHAR ---
124         // Redesenhamos o jogo completo a cada frame
125         desenharTabuleiro(renderer, mouseX);
126 
127         if (fonteGrande && fontePequena) {
128             desenharInterface(renderer, fonteGrande, fontePequena);
129         }
130 
131         // Mostra o frame desenhado na tela
132         SDL_RenderPresent(renderer);
133 
134         // Pausa de 16ms para manter ~60 FPS e nao travar o computador
135         SDL_Delay(16);
136     }
137     if (fonteGrande)  TTF_CloseFont(fonteGrande);
138     if (fontePequena) TTF_CloseFont(fontePequena);
139     SDL_DestroyRenderer(renderer);
140     SDL_DestroyWindow(janela);
141     TTF_Quit();
142     SDL_Quit();
143 
144     return 0;  // Retornar 0 no main() significa que tudo correu bem!
145 }
```

Fonte: Autoria própria.

7.9 Declaração de uso de IA generativa

☐ NÃO utilizei ferramentas de IA generativa para produzir, reescrever ou expandir trechos do texto do meu mini-curso. Eventuais correções automáticas de ortografia/gramática do editor de texto não se enquadram nesta declaração.

☑ UTILIZEI ferramentas de IA generativa de forma ética e transparente, exclusivamente como apoio redacional (ex.: revisão gramatical, melhoria de clareza, sugestão de organização), sem terceirização do conteúdo intelectual. Declaro que: (a) informei o uso; (b) revisei integralmente e sou responsável(a) pelo texto final; (c) conferi a veracidade de dados, citações e referências, garantindo que não há referências inventadas, trechos sem fonte ou conteúdo não verificável.

Se marcou acima a opção UTILIZEI, descreva de forma objetiva:

- **Ferramenta(s)/versão(ões):** Claude Sonnet 4.6;
- **Seções/trechos impactados:** SDL; Conceitos Básicos de Programação em C++; Código Completo do Connect 4;
- **Finalidade(s):** Para dar uma melhor estrutura e ter algoritmos funcionais.

7.10 Bibliografia

Community, S. (2025). Simple directmedia layer. https://wiki.libsdl.org/SDL2/FrontPage. Acessado em: 21 de maio de 2026.

Hasbro (2014). Connect 4 - official game rules. https://instructions.hasbro.com/en-my/instruction/connect-4-game. Acessado em: 21 de maio de 2026.

Microsoft (2026). Documentation for visual studio code. https://code.visualstudio.com/docs. Acessado em: 21 de maio de 2026.

Soares, P. M. S. (2020). Uma metodologia para o desenvolvimento de um agente jogador de connect-4. *IFCE - Campus Aracati*.